Exploring a Striped XML World

Savvas Makalias

Doctor of Philosophy

Laboratory for Foundations of Computer Science

School of Informatics

University of Edinburgh

2010

Abstract

E**X**tensible **M**arkup **L**anguage, XML, was designed as a markup language for structuring, storing and transporting data on the World Wide Web. The focus of XML is on data content; arbitrary markup is used to describe data. This versatile, self-describing data representation has established XML as the universal data format and the de facto standard for information exchange on the Web. This has gradually given rise to the need for efficient storage and querying of large XML repositories. To that end, we propose a new model for building a native XML store which is based on a generalisation of vertical decomposition. Nodes of a document satisfying the same label-path, are extracted and stored together in a single container, a *Stripe*. Stripes make use of a labelling scheme allowing us to maintain full structural information. Over this new representation, we introduce various evaluation techniques, which allow us to handle a large fragment of XPath 2.0. We also focus on the optimisation opportunities that arise from our decomposition model during any query evaluation phase. During query validation, we present an input minimisation process that exploits the proposed model for identifying input that is only relevant to the given query, in terms of Stripes. We also define query equivalence rules for query rewriting over our proposed model. Finally, during query optimisation, we deal with whether and under which circumstances certain evaluation algorithms can be replaced by others having lower I/O and/or CPU cost. We propose three storage schemes under our general decomposition technique. The schemes differ in the compression method imposed on the structural part of the XML document. The first storage scheme imposes no compression. The second storage scheme exploits structural regularities of the document to minimise storage and, thus, I/O cost during query evaluation. Finally, the third storage scheme performs structure-agnostic compression of the document structure which results in minimised storage, regardless the actual XML structure. We experiment on XML repositories of varying size, recursion and structural regularity. We consider query input size, execution plan size and query response time as metrics for our experimental results. We process query workloads by applying each of the proposed optimisations in isolation and then all of their combinations. In addition, we apply the same execution pipeline for all proposed storage schemes. As a reference to our proposed query evaluation pipeline, we use the current state-of-the-art system for XML query processing. Our results demonstrate that:

- Our proposed data model provides the infrastructure for efficiently selecting the

parts of the document that are relevant to a given query.

- The application of query rewriting, combined with input minimisation, reduces query input size as well as the number of physical operators used. In addition, when evaluation algorithms are specialised to the decomposition method, query response time is further reduced.

- Query evaluation performance is largely affected by the storage schemes, which are closely related to the structural properties of the data. The achieved compression ratio greatly affects storage size and therefore, query response times.

Acknowledgements

It is a great pleasure to have the opportunity to thank those who made this research possible. First and foremost, I am heartily thankful to my supervisor, Dr. Stratis Viglas for his continuing encouragement, his valuable suggestions and most importantly his never-ending patience during this work. I deeply value his dedication to all of his students. His professionalism, combined with his high academic level, guided me in the most proper way throughout my research.

I would also like to express my deepest gratitude to Prof. Peter Buneman who gave me the opportunity to work within a top-quality group of fine academics and researchers. Furthermore, I am deeply indebted to all my colleagues at the University of Edinburgh Database Group, who have provided the environment for sharing their experiences about the problem issues involved. In particular, I would like to thank Floris Geerts, Irini Fundulaki and especially Anastasios Kementsietsidis, as in addition to their valuable advice and insights, they made everyday work in the lab enjoyable with their presence and their excellent sense of humour.

Throughout my Ph.D. study, I was lucky enough to meet many interesting people. I especially wish to thank Manos Glynos, Kostas Kiritsis, Ioannis Koltsidas, Kostas Krikelas, Iro Fourtouna, Georgia Papakleovoulou, Gerasimos Skouvaklis, Georgia Vakalopoulou, Apostolos Apostolidis, Kostas Economou, Ifigenia Oikonomopoulou, Panagiotis Douvaras, Niki Athanasiadou, Deri Koutsaki, Sofia Pediaditaki, Anastasia Kostogianou, Arkotong Longkumer, Lindsay Graham, Andrea Bertocco, Lena Wollan, Kifah Hanna and Lise Sorensen for their support and for being there as true friends during this important stage of my life. Their support in this effort is greatly appreciated. A special thanks to Panagiotis, Gerasimos and Kostantis for their hospitality during my last months in Edinburgh and especially to Apostolis who was kind enough to live without his study-room for two months while writing my thesis.

I also want to thank my family for their love and constant support all these years that I have been abroad. In addition, I would like to include my gratitude to my partner for her endless support, understanding and encouragement through the last months of my studies.

Lastly, I would like to thank the Greek State Scholarship Foundation (IKY), for providing me with a Ph.D. scholarship. his study would not have been possible without this financial support.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Savvas Makalias*)

This thesis is dedicated to my beloved father.

Table of Contents

Chapter 1

Introduction

E**X**tensible **M**arkup **L**anguage, XML, is a markup language for structuring arbitrary information. The focus of XML is on data content, *i.e.,* arbitrary markup is used to describe data, as opposed to HTML, a markup language which uses predefined markup to describe data presentation. As a consequence, with XML it is possible to describe any application domain by using tailor-made markup. This versatile, self-describing data representation has established XML as the de facto standard for structuring, storing and transferring data on the World Wide Web.

With the growth of web-oriented services, such as e-commerce and digital libraries, data from various data sources had to be published in a way that was both comprehensible and manageable by these services. The wide acceptance of XML as a universal data format on the web, served that purpose. To that end, significant effort has been expensed into publishing relational data in XML [90, 13, 37], since the majority of application data was, and still is, stored in traditional relational databases. In time, this resulted in a remarkable growth of XML data and efficient ways of storing and querying large XML repositories had to emerge.

There are two options towards this direction. The first is to transform from the XML format to a data format that already know how to process efficiently, for instance the relational data format. The second option is to query directly the XML data, *i.e.,* build a native XML store. In this thesis, we choose the native XML store option and describe a general storage model for efficiently storing and querying large XML repositories.

1.1 Background

Several efforts have been made since the emergence of XML for storing and querying XML data. We now review some of the basic approaches and discuss their impact on query evaluation performance. In Section 1.1.1 we provide background information regarding the conceptual XML model, while in Sections 1.1.2 and 1.1.3, we present approaches that respectively use a relational database or provide native support for XML storage.

1.1.1 Preliminaries

1.1.1.1 Data Model

XML documents have a hierarchical structure and it is thus common practice to model them as trees. To that end, an XML document is modelled as a node-labelled, ordered tree. Each internal node corresponds to an XML element or attribute node and is labelled by the element or attribute name. A leaf node always corresponds to a value node, *i.e.,* attribute value or element content. A tree edge captures the parent-child relationship between any two XML nodes. Another approach for modelling XML documents, is to use an edge-labelled tree; the tree edges and thus node relationships are now labelled instead of the tree nodes. The two models are equivalent in that they both effectively capture the node relationships. We can obtain the edge-labelled tree from the node-labelled tree by "moving" the label of a node to its incoming edge.

When ID/IDREF relationships need to be explicitly captured, the basic tree model must be extended to directed, acyclic graphs (DAG). Graph-based models were also adopted for modelling semi-structured data [6, 45, 7]. This term usually describes data that are schema-less and self-described, *i.e.,* there is no explicit schema to describe the data structure [7]. In that sense, XML is a special case of semi-structured data.

Nodes of the tree/graph data model are assigned a unique id (oid). Thus, for serialising a specific node, storing its oid value is sufficient. Likewise, an edge can be addressed using a pair of oids: $(source, target)$. When order is important (for the tree data model), an *ordinal* value is also needed for capturing edge local ordering. Alternatively, ordering can be retained if instead of assigning random, unique oids to tree nodes, these are generated according to the depth-first traversal of the tree. This way, apart from ensuring uniqueness, oids will also reflect document order.

1.1.1.2 Labelling Schemes

As already described, a tree edge in the XML data model captures a parent-child node relationship. Thus, for identifying whether two nodes a and b, satisfy a parent-child relationship, there must exist a tree edge from a to b. Generalising this, to identify whether two nodes a and b, satisfy an ancestor-descendant relationship, there must exist a path, *i.e.,* a sequence of tree edges starting from node a and reaching node b. In other words, node relationships can be defined by navigating the tree structure. However, in many cases, navigation is inefficient, since the identification of a node relationship, or else a *structural relationship*, may involve the traversal of a large part of the XML tree.

To overcome this limitation, various labelling schemes have been proposed for serialising an XML tree that apart from retaining document order, they enable the identification of structural relationships in constant time. One of the most commonly used category of labelling schemes is the *interval encoding*. The origins of the interval encoding lie in [35], where the tree traversal order is used for identifying ancestor-descendant tree node relationships. According to [35]:

> "For two given nodes x and y of a tree T, x is an ancestor of y if and only if x occurs before y in the preorder traversal of T and after y in the postorder traversal".

This gave birth to the *PrePost* labelling scheme of the interval encoding category, according to which:

$$y \text{ is a descendant of } x \Leftrightarrow x.pre < y.pre \wedge y.post < x.post$$

where a $pre(post)$ value corresponds to the value assigned to a node during the tree pre(post)order traversal.

A similar scheme to the PrePost labelling was also proposed in [105]. Borrowing ideas from inverted list text indexing, the authors propose the *region encoding* for labelling tree nodes. In its simplest form, each node is assigned a pair of $(start, end)$ values; the $start$ value is assigned as a node is visited during a depth-first traversal of the tree[1], while the end value is assigned upon leave during the traversal (*i.e.,* after all of its children nodes are also visited). We now have that:

$$y \text{ is a descendant of } x \Leftrightarrow x.start < y.start < y.end < x.end$$

[1] The $start$ value of a node is the same as its preorder rank: pre.

which can be reduced to:

$$y \text{ is a descendant of } x \Leftrightarrow x.start < y.start < x.end$$

since $y.start < y.end$ always holds by definition.

A variant of the region encoding is the *extended preorder traversal* labelling scheme (PreSize), proposed in [63]. Each node is labelled with a pair of $(order, size)$ values, where *order* is the node's preorder rank and *size* can be any number larger than the number of its descendant nodes. Similar to the other interval encoding labelling schemes described above, we have that

$$y \text{ is a descendant of } x \Leftrightarrow x.order < y.order < x.order + x.size$$

This scheme was proposed to accommodate a number of future insertions of XML nodes, without the need of re-numbering. It is easy to see that if *size* is equal to the number of the node's descendant nodes, the PreSize scheme becomes the region encoding scheme.

Any of the three proposed labelling schemes of the interval encoding category can be enriched with the *level* of a node, to support tight containment (parent-child) property in addition to containment (ancestor-descendant) property. The *level* of a node is the number of edges that connect the tree root with the node. For any of the interval labelling scheme, we now have that:

$$y \text{ is a child of } x \Leftrightarrow (y \text{ is a descendant of } x) \wedge y.level = x.level + 1$$

Finally, in [47], a variant of the PrePost scheme is used, where each node is labelled by a $(pre, post, par)$ triplet, where *par* is the parent's preorder rank. This labelling scheme effectively enables the identification of any possible node relationships as these are defined in XPath specification [32, 14]. Similar variants can be used for the PreSize or the region encoding schemes. All three flavours of interval encoding are equivalent in that node relationships can be identified by using simple, arithmetic operations on the node's structural information, captured by the labelling scheme.

Another category of labelling schemes that has received much attention is that of the *prefix-based* labelling schemes. In this category, the label of a node u encodes all nodes on the path from the tree root to node u. In general,

$$y \text{ is a descendant of } x \Leftrightarrow \mathsf{label}(x) \text{ is a prefix of } \mathsf{label}(y)$$

$$y \text{ is a child of } x \Leftrightarrow \mathsf{label}(x) \text{ is the maximal prefix of } \mathsf{label}(y)$$

Examples of prefix-based labelling schemes is the *Dewey Order* [95] and *ORDPATHs* [82]. A recent survey of XML labelling schemes can be found in [53].

1.1.2 The Relational Approach

Since the very first years that XML gained popularity, many database researchers considered leveraging relational database technology for storing XML data and providing efficient querying mechanisms. The reasoning was that we should take advantage of the experience gained over the last 30 years of research in this area. In this section we present a few of the proposed approaches for storing XML data in relational databases. These fall into two main categories depending whether the structure of the XML data is known a priori or not.

1.1.2.1 Schema Aware Decomposition

One of the first approaches to store XML data that conforms to a schema (DTD) in a relational database, was proposed in [91]. The authors investigated the correspondence between elements and attributes of DTDs and entities and attributes of the ER-model. They claimed that a direct mapping of DTD elements to relations would possibly lead to excessive fragmentation of the document and proposed three *inlining* techniques for mapping a DTD into a relational schema. The key ideas were: (a) a relation is created for a DTD element that also includes as many of its descendant sub-elements as possible, and (b) all set-valued and recursive elements are stored in separate relations. The proposed techniques differ in the degree of redundancy, for instance an element can be stored in a single or multiple relations. The basic inlining technique, which presents a high degree of redundant elements, results in reducing the number of join operations during query evaluation. However, it was found to be impractical in most cases due to the large number of produced relations. The other proposed inlining techniques trade storage cost over query performance.

Another proposal for mapping XML data to a relational schema is a mapping engine, *LegoDB* [15]. This time, a cost-based approach is considered; LegoDB exploits an XML schema which is also enhanced with data statistics [41], to construct a search space of possible relational mappings. It then selects the best option based on a given query workload. An important difference with respect to the inlining approach, is that the latter simplifies the input DTD to an equivalent one that can be easily mapped into a relational schema according to the inlining technique. In LegoDB, schema transformations are also applied but only to produce alternative mappings. This creates a search space of relational mappings which is then explored according to the estimated performance of each mapping for a certain query workload.

Both of the approaches described above rely on the source XML schema to derive the target relational schema. The inlining approach only depends on the available DTD information. LegoDB, in addition, utilises statistics and a query workload to choose among the alternative mappings. Thus, by inlining or outlining of elements and attributes, it can choose a mapping that groups together XML data that are usually accessed together. In conclusion, the inlining technique is a more generic approach while the LegoDB approach is application-specific.

1.1.2.2 Schema Oblivious Decomposition

Although it is common for XML documents to conform to a specific schema, this does not always happen. In addition, it is often the case that a schema becomes old and documents that are updated to support new features (in a web service for instance) are invalidated. To support these cases, schema-oblivious mappings were introduced that usually employ a fixed relational mapping in contrast to the schema-driven approaches described above.

The *STORED* approach [33], generates a mapping between the semi-structured data model and the relational data model. STORED is the only exception of the schema-oblivious approaches that do not employ a fixed mapping. Instead, a relational mapping is produced based on data-mining techniques which examine the semi-structured data instance and identify tree patterns. Apart from the relational mapping that is selected for storing the semi-structured data, an overflow graph is also defined that is used for storing parts of the source data that are not mapped in the produced mapping.

A schema-oblivious approach that is based on a fixed relational schema is that of [40]. In that work, the main idea is to study simple ad-hoc mappings that will later act as the baseline cases for future, sophisticated approaches. The first of the proposed mappings is the *Edge* approach, where the document structure is stored in a single table, the Edge table. As its name suggests, each tuple in the Edge table corresponds to a graph edge, *i.e.,* a parent-child node relationship and contains the $(source, target)$ oids of the connected nodes. The structure of the Edge table is:

$$Edge(source, ordinal, label, flag, target)$$

where *label* is the element tag name of the target node, and *ordinal* corresponds to the local ordering of an edge. Finally, *flag* indicates whether the target node is an internal or leaf node. Another mapping proposal in [40] is the *Binary* approach, which

is essentially a horizontal partitioning of the Edge table on the *label* attribute. Tuples with different *label* values are now stored in separate tables. The structure of a Binary table is:

$$Binary_{label}(source, ordinal, flag, target)$$

The number of the produced tables is the number of unique labels *i.e.,* element tags, that occur in the XML document. Finally, the *Universal* approach is the full outer join of all Binary tables. The structure of the Universal table is:

$$Universal(source, ordinal_{l_1}, flag_{l_1}, target_{l_1}, ordinal_{l_2}, flag_{l_2}, target_{l_2}, \ldots, ordinal_{l_n}, flag_{l_n}, target_{l_n})$$

where $l_1, l_2, \ldots, l_n$ are label names. The Universal table is the denormalised approach and as a result contains many null and redundant values. Textual values can either be stored in a separate *Values* table or alternatively be inlined into the basic tables (this applies to all three approaches). During query evaluation, selection of XML nodes is handled by join operations on the Edge (Binary) relation(s). For complex path expressions that include a large number of parent-child relationships, the Edge approach underperforms in most cases as it employs a large number of self-join operators on the large Edge table for selecting the appropriate XML nodes. The Binary approach performs better for most cases since the partitioning allows the selection of relevant tables and thus significantly reduces query input.

We described the Binary approach as the horizontal partitioning of the Edge table on the *label* attribute. Another approach for mapping XML data to relational data is the horizontal partitioning of the Edge table at the rooted label-path of the target node. To that end, tuples with different label-paths are now stored in separate tables. The *Monet* XML model [87] defines a number of binary relations, each for a unique rooted label-path that appears in the XML document. All tree edges that comprise the last part of a label-path starting from the root of the document are stored together in a binary relation as a $(source, target)$ oid pair that is associated with the lasting edge in the label-path. The Monet model is a refinement not only of the Edge approach but of the Binary approach as well, accomplishing a higher degree of fragmentation. This approach comes in contrast to the claim in [91] that a high degree of fragmentation should be avoided to avoid excessive join operations. Nevertheless, the authors in [87] argue that their approach is efficient since only small amounts of data are involved in these joins operations.

Another work that uses a fixed relational mapping for XML storage is *XRel* [104]. In contrast to the edge-oriented approaches described above, XRel adopts a node-oriented approach for XML storage, serialising XML tree nodes as relational tuples. The key ideas of XRel are: (a) tree nodes are decomposed into separate relations according to their type, (b) nodes are stored along with their (encoded) label-path, and (c) each node is assigned a *region*, a pair of $(start, end)$ positions which resembles the region encoding (see Section 1.1.1.2). The XRel mapping is comprised of the following four relations:

$$Element(docID, pathID, start, end, index, reindex)$$
$$Attribute(docID, pathID, start, end, value)$$
$$Text(docID, pathID, start, end, value)$$
$$Path(pathID, path)$$

where *docID*, *pathID* are document and label-path identifiers, *start* and *end* attributes shape a document region while *index* and *reindex* attributes represent the order of an element node among their siblings in document order and reverse document order respectively. *Attribute* and *Text* relations also contain a *value* attribute which stores the attribute or text value respectively. Finally, label-paths (*path*) are explicitly stored in the *Path* relation. Selection of XML nodes during query processing is handled by the evaluation of regular expressions on the label-path attribute of the *Path* relation (in particular using the SQL operator LIKE) and a join operator between the relation containing nodes of the desired type and the *Path* relation.

A similar relational mapping is that of *XParent* [58]. In this work, however, the edge-oriented storage approach is adopted. The fixed mapping is as follows:

$$LabelPath(pathID, len, path)$$
$$DataPath(Pid, Cid)$$
$$Element(pathID, id, ordinal)$$
$$Data(pathID, id, ordinal, value)$$

where the relation *LabelPath* acts as the *Path* relation of XRel, with the exception that it also stores the length of the label-path. Relations *Element* and *Data* hold structural and content XML nodes respectively as their name suggests. The *id* attribute corresponds to the target node's unique oid that occurs on label-path of *pathID*, while *ordinal* and *value* attributes are self-explanatory. Finally the *DataPath* relation is a

Parent relation which stores parent-child edges (Pid, Cid stand for parent-id, child-id). As in the case of the Edge approach, to evaluate ancestor-descendant relationships, a large number of join operations is required. To overcome this, XParent materialises an Ancestor relation which explicitly stores all ancestor-descendant relationships. Ancestor information is redundant and it only serves to accelerate the evaluation of ancestor-descendant relationships trading performance efficiency for storage cost. During query processing, XParent acts as a combination of the Edge approach and XRel. By using path-labels and the Ancestor relation, it can effectively locate XML nodes that are either related to a certain label-path or take part in an ancestor-descendant relationship, avoiding a large number of join operations. In addition, the use of equi-joins for traversing short path expressions and locating values is usually more efficient than the evaluation of θ-joins that occur in XRel for testing document regions.

XPath Accelerator [47] is another proposal for a schema-oblivious XML-to-relational storage mapping. As its name suggests, it serves as an indexing scheme for accelerating the evaluation of XPath expressions. It is one of the few proposals that considers the evaluation of *all* XPath axes, in addition to the child and descendant axis which was the focus of the majority of existing work. XPath Accelerator uses a node-oriented relational mapping, storing all document nodes in a single table, the *Accel* table. To serialise the document nodes and preserve the structural relationships between them, is uses the PrePost labelling scheme. To that end, for efficiently supporting navigation for all XPath axes, an extended PrePost labelling scheme is used of the form of $(pre, post, par)$ triplets, as described in Section 1.1.1.2. The structure of the Accel table is:

$$Accel(pre, post, par, att, label)$$

The *att* flag is used to indicate whether a node is an attribute node or not. Similarly to the relational mappings proposed in [40], content can be stored in a separate table $Data(pre, text)$ or it can be inlined in the Accel table. In the latter case, the Accel table becomes:

$$Accel(pre, post, par, kind, label, text)$$

where *att* flag is now replaced by the *kind* attribute denoting the node kind and *text* attribute stores element content/attribute values. The authors, based on the observation that the four main XPath axes *i.e.,* descendant, ancestor, following and preceding, specify a document partitioning for any node, they have defined specialised XPath axes evaluation conditions, called axes windows. When generating an SQL query for a lo-

cation step, these conditions are translated to a θ-join on the node PrePost attributes $(pre, post, par)$. A sequence of locations steps is translated recursively, with the (subquery) result of one location step expression, providing the input for the evaluation of the next one. The translation scheme of an XPath expression of n location steps generates an SQL query of nesting depth n. However, this can be flattened into an n-ary self-join. When only the child and descendant axes are involved, XRel, XParent and Monet approaches may perform better since the usage of label-path information will reduce the n number of join operations needed. However, this is only restricted to these two axis, while XPath Accelerator provides full XPath axes support. In addition, the true power of the XPath Accelerator lies in the axes window queries that in combination with efficient access methods can restrict access to the Accel table, only to the candidate nodes that are relevant to a given axis.

1.1.3 The Native Approach

In addition to providing support for XML (and semi-structured data in general) by using traditional relational technology, many researchers tried to tackle the same problem from a different angle. They believed that native systems should be developed to reflect the structure and properties of the XML data.

1.1.3.1 Structural Summaries

Early work in this direction introduced a special structure (usually a DAG) serving as a *structural summary* of the XML data source. A structural summary of a data graph G is, as its name suggests, a compact graph structure G' that summarises the structure of the original data graph G. Each node in G' corresponds to a set of data nodes in G, called its *extent*. The first proposal for a structural summary is that of a *DataGuide* [45], introduced as part of a native database management system for semi-structured data, *Lore* [71]. The idea of a DataGuide was to provide a concise and accurate path summary of the data graph that would assist query formulation and optimisation. It is concise in the sense that it contains each unique label-path of the data graph exactly once and accurate because all label-paths it contains, also exist in the data graph. For each data graph there exist multiple DataGuides. One could argue that the minimal DataGuide is always the best choice. This does not always hold. Apart from a path summary, a DataGuide can further assist query optimisation by storing data samples or other statistical information. This, however, is meaningless

in practice since multiple label-paths can lead to the same DataGuide node, even if the label paths have different extents in the data graph. To that end, the *strong DataGuide* was proposed so that if label-paths l_1 and l_2 both reach the same node in G', then l_1 and l_2 have the same extent in G. The downside of a strong DataGuide is that its size can be, in the worst case, exponential of the size of the data graph.

To overcome the problem of a possibly large structural summary, other graph-based structures have been proposed that are based on the following key idea: The nodes of data graph in G are grouped together into equivalence classes and a summary node is created for each of the classes. Then, if there exists an edge from any of the nodes in the extent of the equivalence class a to any node in the extent of the equivalence class b, a summary edge is also added that connects the summary nodes corresponding to equivalence classes a and b. Such a structural summary is the 1-index [74], using *backward bisimilarity* for partitioning data graph nodes into equivalence classes. In detail, two data nodes are grouped together only if they share the same incoming rooted paths, *i.e.,* the same label-paths. The size of the 1-index is at most equal to the size of the data graph G. To further reduce the size of the 1-index of a data graph, the notion of *backward k-bisimilarity* was introduced in [61], based on which a family of approximate structural summaries were built, $A(k)$-indexes. The idea of k-bisimilarity is that it constrains the data nodes that are partitioned on a notion of a local structure, considering incoming paths of length up to k. Parameter k trades the index's accuracy for its size. When k is large enough to cover the largest label-path in the data graph, the $A(k)$-index becomes the 1-index and is therefore precise for any path expression. For smaller values of k, though, the $A(k)$-index's size is reduced but is now approximate for path expressions involving paths of length larger than k; for those queries, a post-validation process is required for eliminating false positive results.

All structural summaries discussed so far can cover simple path expressions. For evaluating branching queries, the $F\&B$-index has been proposed in [7]. $F\&B$-index partitions data nodes using forward and backward bisimilarity and thus considers not only the incoming paths of the data nodes but their outgoing paths as well. It is shown that the $F\&B$-index is the smallest index that covers any branching query [60]. Nevertheless, it can become as large as the data graph itself, rendering it impractical. To that end, in the same spirit of the $A(k)$-index, which was proposed as an approximate index that covers simple path expressions and reduces the size of 1-index, the authors in [60], propose a family of $(F+B)^k$-indexes. Similarly to the $A(k)$-index, the k parameter is used to trade accuracy over size. For a small value of k, the size of the $(F+B)^k$-index is

significantly reduced compared to *F&B*-index but it can provide accurate results only for a subclass of branching queries.

Another approach of a structural summary is that of the *skeleton* in [20]. The authors propose a compression technique for creating a main memory summary of the XML tree structure. Nevertheless, instead of building a path index for covering a class of simple or complex path expressions, the skeleton is proposed as a compressed form of the XML tree structure that renders the original tree useless. Compression only applies to the XML structure, ignoring all textual values and is based on *sharing common subtrees* technique: all tree nodes are partitioned into equivalence classes based on forward bisimilarity, *i.e.,* considering the outgoing paths of the data nodes. An important feature of the skeleton approach is that unlike all structural summaries described above, the order of the outgoing edges is significant; they implicitly retain order. This renders the compression technique as lossless and thus: (a) query expressions can be evaluated directly on the compressed instance, and (b) the original XML tree can be fully reconstructed. This effectively renders the original XML tree useless.

1.1.3.2 Native XML Systems

We now describe the architecture of some of the most important native systems for storing and querying XML data (and semi-structured data in general).

The *Lore* project [71] is one of the first native approaches for managing semi-structured data. Lore stands for "Lightweight Object REpository" and uses an object model (OEM), which is a graph-based model, to describe semi-structured data. Objects are physically stored in disk pages. Objects are of variable size; many objects may be stored in a single page. When an object grows (due to update operations) and can no longer be accommodated in the same page, it will be moved to another page. In addition, Lore's physical storage model permits large objects to span between multiple pages. For enabling efficient navigational access, objects in Lore are clustered in a page based on the depth-first order. However, since an object may have multiple parent objects (graph model), this is not always possible. In this case, it is clustered along with one (arbitrary) of its parent's object. The basic Lore physical operator, the *Scan* operator, is used for navigation traversal. In addition, Lore employs a large variety of indexes for the efficient selection of objects. Due to the lack of an explicit schema, Lore uses the strong DataGuide [45] to infer the database structure and optimise queries. The DataGuide acts as a path index (Pindex) for locating all objects that are reachable by a certain path. In addition, value indexes (Vindex) are used for locating objects satis-

fying a value-based predicate and (optionally) a given label[2]. Link indexes (Lindex) are also employed for locating all parent objects of a given object, that satisfy a given label. Finally, edge indexes (Bindex) are used for locating all parent-child object pairs connected via a given label. A detailed description of all Lore indexes can be found in [73].

Lore was proposed for storing and managing semi-structured data. Most approaches, however, focus on XML data. *Natix* [39, 59] was proposed as an XML database management system. Similarly to Lore, the Natix storage manager clusters subtrees of an XML document tree into physical records which are stored in disk pages. This implies that each record and thus subtree must fit in a page. However, the entire XML tree can rarely fit in a single page. To waive this restriction, the logical tree is semantically partitioned into subtrees, so that each of them can be stored as a physical record on a single page. For enabling backward navigation, each record also stores a pointer to the physical record containing the parent node of the root node of its subtree. The Natix storage architecture preserves the original tree structure. In addition to tree node navigation, Natix also provides two types of indexes: inverted-list style indexes for efficient text indexing and a structural index, that using the region encoding scheme, stores the serialised node edges (similar to the parent relation in XParent).

Timber [55] is another native XML database management system in which XML data is stored directly in its natural tree structure. The data manager stores an internal representation of the document tree on top of the SHORE [22] storage manager. A data manager node corresponds to an element and child nodes are added for any of its sub-elements. In addition, attributes are all gathered together and stored as a single child node of the element node. The element content is also stored as a separate, child node. For the efficient processing of parent-child and ancestor-descendant node relationships, nodes are also enriched with region encoding labels. Query evaluation in Timber is largely dominated by structural join operations, and the authors have proposed efficient algorithms for evaluating those [8]. These, however, operate on ordered lists of nodes, *i.e.,* lists of nodes that occur in document order and usually reflect to all nodes of a specified label (tag). For the efficient identification of such lists of nodes, Timber employs B^+-tree indexes on tag labels. In addition, for processing value-based predicates efficiently, Timber also employs value indexes on both attribute values and element content while in the case of long textual values, search term inverted-indexes are preferred.

[2]Reachable by an edge of a certain label.

XISS [63], which stands for XML Indexing and Storing System, is a native XML store that was developed to efficiently support regular path expressions. The authors proposed the PreSize labelling scheme for serialising document nodes while retaining their tree structure information. The proposed labelling scheme in conjunction with the proposed merge-based structural join algorithms, enable the efficient processing of ancestor-descendant node relationships. For supporting these operations, three major index structures were proposed: The element, attribute and structure indexes. The element and attribute indexes provide access to elements and attributes respectively of a specific label and document. They are both implemented as B^+-trees; each entry in a leaf node points to a set of fixed-length records for elements (attributes) having the requested label and grouped by the document they belong to. Each element record includes the $(pre, size)$ pair and other element related information, such as the element $level$. Element records are ordered by the *pre* values. The attribute record is similar to the element record, only that it also includes the id of the attribute value[3]. Finally, the structure index, provides an array of all element and attribute nodes for each of the XML documents stored at the XISS repository. Each array is sorted on the *pre* values. Each record, apart from the $(pre, size)$ pair, it contains parent and children information to enable tree navigation. The proposed index scheme can support many different retrieval operations. However, unlike Timber, that builds unclustered indexes on top of tree-style document representation, XISS uses indexes which are clustered on their search attributes. This results to more efficient access methods but also implies significant data redundancy.

A similar approach was considered by the *Niagara* project [77, 51], where a mixed mode evaluation process is considered. In detail, to support both navigational processing as well as structural joins and interchange processing paradigms during the query execution, a specialised storage scheme was proposed that includes two modules: the data manager and the index manager. Both storage modules use the region encoding ($(start, end)$ pairs) for document node serialisation. The data manager is implemented as a B^+-tree and provides the tree-based representation of an XML document. Using the B^+-tree key attributes, *i.e.,* $(docID, start)$ values, it is possible to retrieve full structural information of an XML document element, in addition to its children list in document order. On the other hand, the index manager is a two level index that provides a list (posting list) of all nodes with a specified label on a specified document. This is again implemented as a B^+-tree, with $(label, docID)$ acting as keys. When the posting list is

[3]In XISS, all content values are assigned a value id and stored in a value table.

very small, it is kept on the B$^+$-tree leaf page. Otherwise, a second level index is built for each posting list and a separate B$^+$-tree index is used. We observe that the architecture of the index manager is much more complex compared to the simple B$^+$-tree element indexes of XISS. Both systems can efficiently retrieve all nodes of a specified label. However, the index manager can additionally access a specific part of them, using the second level index.

1.2 Motivation

We are interested in efficient techniques for storing and querying large XML repositories. As already described, many approaches have been proposed for managing semi-structured and XML data. We believe that for managing large data repositories a data management system must satisfy the following conditions:

1. Schema independence
2. Permanent storage

We strive for schema independence as most XML documents found "in the wild" are not bound to conform to a schema. In addition, even if there exists a schema that an XML document conforms to, it is normal to assume that as it evolves over time, new or updated data may not conform to a static schema, especially if data is retrieved from various sources. It is thus critical for a system that stores and manages large amounts of XML data, to be schema oblivious. To that end, we depart from schema-conscious approaches like [91, 15], which propose a mapping for XML storage that depends on priori knowledge of a schema (DTD or XML Schema).

Furthermore, the implementation of an XML store that is expected to manage big volumes of data, needs to be immune to potential main memory limitations and thus rely solely on storing data on persistent storage. As a consequence, we also depart from the structural summaries approach. As described in Section 1.1.3.1, the size of a structural summary that faithfully maintains the document structure can be large; the DataGuide may be exponentially larger than the original document while the 1-index and *F&B*-index can get as large as the original document. The compressed skeleton proposed in [20] can effectively reduce the size of the structure of a regular XML document; it is questionable though if the compressed skeleton will always fit in main memory, especially in the case of large, irregular XML documents. More compact structural summaries were also proposed [60, 61], although these summaries

are index proposals that trade size over precision and cover certain query subclasses. The original structure is also needed.

Several native approaches have emerged that reflect the structure and properties of the XML model. At the physical layer, XML data is stored in its natural tree structure. Nevertheless, for efficiently supporting node relationships, especially ancestor-descendant relationships, these native approaches employ additional indexes or specialised structures that operate on the node's structural properties, encoded by a labelling scheme. Furthermore, in the XParent [58], XRel [104] and Monet [87] approaches, for avoiding expensive navigational-based traversals, label-path information is used (implicitly or explicitly) as a means for locating relevant nodes.

Motivated by the processing paradigms described above, we propose a storage model for XML that is reminiscent of vertical partitioning. We explore the effectiveness of the decomposition technique for storing and querying large XML repositories. By decomposing large fragments of the original structure (and content) into smaller, path-based fragments that share similar properties, it is easier to select parts of the document that are relevant to a given query. Instead of building additional indexes for locating certain parts of the document, the building blocks of the proposed decomposition directly act as indexes on the document structure. Monet [87] has already applied a similar decomposition for enabling efficient query evaluation. Nevertheless, our proposal departs from the navigational-based execution and further exploits the application of already proposed, novel features for efficient query processing over the proposed model. We employ a labelling scheme for serialising the document tree while retaining full structural information. In addition, we consider structural join algorithms for the efficient evaluation of node relationships over our storage model. Finally, we strive for further optimisation opportunities that arise due to the new representation.

1.3 Contributions

We propose a new model which is based on a generalisation of vertical decomposition. Nodes of a document satisfying the same document label-path, are extracted and stored together in a single container, termed a *Stripe*. Over this new representation, we introduce evaluation techniques, which allow us to handle a large fragment of XPath 2.0.

We focus on the optimisation opportunities that arise from our decomposition model during any query evaluation phase. At first, during query validation, we present an input minimisation process that exploits the proposed model for identifying mini-

mal data input, in terms of Stripes, relevant to the given query. We also define query equivalence rules for query rewriting over our proposed model. This effectively reduces a given query, *i.e.,* it replaces query expressions with equivalent expressions (over our striped decomposition), that require less I/O and computation while, at the same time, produce the same result. Finally, during query optimisation, we deal with whether and under which circumstances certain evaluation algorithms can be replaced by others having lower I/O and/or CPU cost. Each of the proposed optimisations, has a significant impact on query performance which may vary depending on the target XML dataset and the tested query.

We also propose three different storage schemes under our general decomposition technique. The schemes differ in the compression method imposed on the structural part of the XML document. The first and most natural storage scheme, the explicit storage scheme, is one where no compression is imposed, and as such, each node in a Stripe corresponds to a single document node. The tree-sharing compression storage scheme exploits structural regularities in the document to minimise storage and, thus, I/O cost during query evaluation. Finally, the agnostic compression storage scheme performs structure-agnostic compression of the document structure which results in minimised storage, regardless of the actual XML structure. For the two storage schemes that employ structural compression, query performance is strongly coupled with the achieved compression; the reduction of the query input size compared to that of the explicit storage scheme, is always reflected in the evaluation results.

We have conducted an experimental study over a set of XML repositories of varying size, recursion and structural regularity. The metrics considered for the experimental results are: (a) query input size, (b) query execution plan size (in terms of number of operators used), and (c) query response time. We process query workloads for all XML repositories by applying each of the proposed optimisations in isolation and then all of their combinations. Our results demonstrate the contribution of each optimisation, as well as their added impact during query evaluation. In addition, we apply the same execution pipeline for all proposed storage schemes and compare the produced results We also compare our results to the current state-of-the-art system for XML query processing, acting as a reference for our proposed query evaluation pipeline. Our results demonstrate that:

- Our proposed data model provides the infrastructure for efficiently selecting the parts of the document that are relevant to a given query.

- The application of query rewriting, combined with input minimisation, reduces query input size as well as the number of physical operators used. In addition, when evaluation algorithms are specialised to the decomposition method, query response time can be further reduced. This is independent of the actual storage scheme being used.

- Query evaluation performance is largely affected by the storage schemes, especially those that impose data compression, which are in turn closely related to the structural properties of the data. The achieved compression ratio (subject to the structural data properties) greatly affects storage size and as a consequence, query response times.

1.4 Thesis Roadmap

The rest of the thesis is organised as follows: In Chapter 2, we define our decomposition model, *Striping*, and introduce optimisations for identifying minimal data input and enhancing query performance. In Chapter 3, we describe the evaluation algorithms and access methods over the new representation, which allow us to handle a large fragment of XPath 2.0. In addition, we explore whether and under which circumstances, these can be enhanced due to the decomposition model. In Chapters 4, 5 and 6 we describe three storage schemes under the general decomposition model and compare their effectiveness in terms of (a) structural compression and (b) query evaluation efficiency. In Chapter 7, we summarise our results.

Chapter 2

Query Reduction over a Striped Model

In this chapter we introduce an XML data model which aims to provide a query engine with the minimum data necessary in order to evaluate XML queries. We employ a partitioning decomposition that is reminiscent of vertical partitioning in relational database systems [12, 31], which we term *Striping*. We define *Stripes* as the building block of our decomposition model and identify opportunities regarding: (a) minimising data input with respect to a given query, and (b) minimising the query itself in terms of query terms.

In relational database systems, the SQL compiler is responsible (among many things) for parsing an SQL query and identifying all relations and their corresponding attributes that contain all data relevant to the query. To perform such validity checks, it uses database metadata stored in the system catalog. In our striped representation, we use Stripes as the building block to store XML data, in analogy to relations that are used for storing data in a relational database system. We present a process for identifying data input which is relevant to a given query. We then exploit the proposed model in order to reduce that, if possible. We term this *Input Minimisation*. In addition, we provide query equivalence rules, defined over the proposed model that reduce a given query *i.e.,* replace parts with equivalent expressions that require less computation and, at the same time, produce the same result. We term this *Path Expression Minimisation*.

The rest of this chapter is organised as follows: In Section 2.1, we review a general XML model commonly used and provide a subset of the XPath language we consider for the purpose of query reduction along with its formal semantics. In Section 2.2, we propose a *striped* data model for storing and querying XML data, based on a general decomposition method. In Section 2.3, we focus on Stripe processing as a means to minimise the input of an XML query in terms of Stripes, while in Section 2.4, we define

path expression equivalences that hold over the proposed data model and exploit them in order to reduce unnecessary path expression operations.

2.1 Preliminaries

2.1.1 XML Data Model

We now present a formal data model for XML. Some definitions are adopted from the formal XML data model as specified in [99]. The basic data type for an XML model is that of a *Node*. Any node (of type *Node*) can be one of the following *kinds*: root, element, attribute or text. Predicate functions isRoot, isElement, isAttribute and isText of type $Node \rightarrow Boolean$ test the kind of a node; for a specific node, only one kind predicate holds.

XML documents have a hierarchical structure and it is therefore common practice to model them as trees. To that end:

Definition 1.1: *An XML document T is a rooted, directed, node-labelled tree $T = (V, E, r, \mathcal{L}_V, \mathsf{oid}, \mathsf{label}, \mathsf{text})$. A tree node $u \in V - \{r\}$ is an XML Node corresponding to an element, an attribute or a text value (CDATA/PCDATA) of the document. Each node is given a unique object id (oid) via function $\mathsf{oid} : Node \rightarrow String$ and a label via function $\mathsf{label} : Node \rightarrow \mathcal{L}_V$. Element and attribute nodes are labelled with element or attribute names. In addition, leaf nodes that correspond to document character values are given the distinguished label "#text" and a value via function $\mathsf{text} : Node \rightarrow String$. Node r is a distinguished node of V, the document root, labelled as "#root". $\mathcal{L}_V$ is the domain of node/attribute labels also containing strings "#root" and "#text". A tree edge $(u \rightarrow v) \in E$ defines a parent/child relationship between any two nodes u, v. Finally, relation $\ll$ of type $Node \times Node \rightarrow Boolean$, defines a total ordering between all nodes which corresponds to the depth-first traversal order of tree T.* □

Note that the root node r of an XML tree T corresponds to the document node as defined in the XPath data model [101]. An example document tree is shown in Figure 2.1.

We have already mentioned the basic data type *Node*. In general, for any type T, $Set(T)$ and $Set_1(T)$ define two new types, the set and singleton set of elements of type T. For instance $Set(Node)$ defines the type of set of Node elements.

As described in Definition 1.1, there exists a primitive relationship between tree

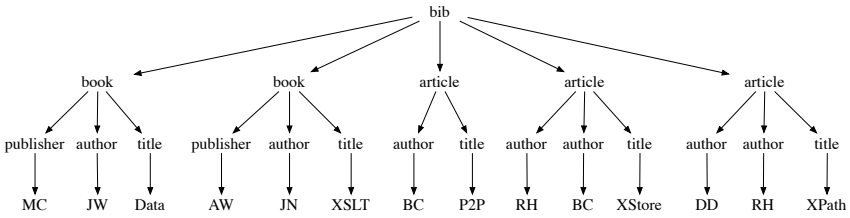


Figure 2.1: An example XML document

nodes (*Nodes*), the parent-child relationship, enforced by a tree edge. We represent a relationship between *Nodes* with a mathematical function $f : Node \rightarrow Set(Node)$. For any function f, its transitive closure f^+ and reflexive and transitive closure f^* are recursively defined as:

$$
\begin{aligned}
f^0(x) &= \{x\} \\
f^n(x) &= \{z \,|\, y \in f^{n-1}(x) \wedge z \in f(y)\} \text{ for any } n \in \mathbb{N} \\
f^+(x) &= \cup_{n \in \mathbb{N}-\{0\}} f^n(x) \\
f^*(x) &= \cup_{n \in \mathbb{N}} f^n(x)
\end{aligned}
$$

There exist four primitive functions that correspond to the basic relationships among nodes: Function $\mathsf{children} : Node \rightarrow Set(Node)$ selects (the set of) children nodes, while function $\mathsf{parent} : Node \rightarrow Set_1(Node)$ selects the (singleton set of) parent node. Function $\mathsf{attributes} : Node \rightarrow Set(Node)$ selects the (set of) attribute nodes and finally, function $\mathsf{root} : Node \rightarrow Node$ selects the document root node of the tree in which a node exists. Further relationships between Nodes can now be defined, based on the primitive functions described above. For instance, function $\mathsf{parent}^+(x)$ (the transitive closure of function parent), defines the relationship between a node x all its ancestor nodes. Similarly, function $\mathsf{children}^*(x)$ defines the relationship between a node x and its descendant nodes (including itself). We now provide the definition of the primitive functions as follows:

$$
\begin{aligned}
\mathsf{children}(x) &= \{y \,|\, (x \rightarrow y) \in E \wedge (\,\mathsf{isElement}(y) \vee \mathsf{isText}(y)\,)\} \\
\mathsf{parent}(x) &= \{y \,|\, (y \rightarrow x) \in E \wedge (\,\mathsf{isElement}(y) \vee \mathsf{isRoot}(y)\,)\} \\
\mathsf{attributes}(x) &= \{y \,|\, (x \rightarrow y) \in E \wedge \mathsf{isAttribute}(y)\} \\
\mathsf{root}(x) &= \{y \,|\, y \in \mathsf{parent}^*(x) \wedge \mathsf{isRoot}(y)\}
\end{aligned}
$$

where E is the set of edges of tree T. Furthermore, we define a sibling relationship by

function siblings : $Node \rightarrow Set(Node)$:

$$\mathsf{siblings}(x) = \{\, z \,|\, y \in \mathsf{parent}(x) \wedge z \in \mathsf{children}(y) \,\}$$

from which the following holds:

$$x \in \mathsf{siblings}(y) \Rightarrow y \in \mathsf{siblings}(x)$$

where $\Rightarrow$ stands for: *implies*. From the definition of the ordering relation $\ll$ (as described in Definition 1.1), we also provide the following properties:

$$x \ll y \Rightarrow \; y \notin \mathsf{parent}^*(x)$$
$$x \ll y \Rightarrow \; x \notin \mathsf{children}^*(y)$$

In addition, we define functions preceding, following, preceding-sibling and following-sibling: $Node \rightarrow Set(Node)$ as follows:

$$\begin{aligned}
\mathsf{preceding}(x) &= \{\, y \,|\, y \in \mathsf{children}^+(\mathsf{root}(x)) \wedge y \notin \mathsf{parent}^+(x) \wedge y \ll x \,\} \\
\mathsf{following}(x) &= \{\, y \,|\, y \in \mathsf{children}^+(\mathsf{root}(x)) \wedge y \notin \mathsf{children}^+(x) \wedge x \ll y \,\} \\
\mathsf{preceding\text{-}sibling}(x) &= \{\, y \,|\, y \in \mathsf{siblings}(x) \wedge y \ll x \,\} \\
\mathsf{following\text{-}sibling}(x) &= \{\, y \,|\, y \in \mathsf{siblings}(x) \wedge x \ll y \,\} \\
x \in \mathsf{preceding}(y) &\Rightarrow y \in \mathsf{following}(x) \\
x \in \mathsf{preceding\text{-}sibling}(y) &\Rightarrow y \in \mathsf{following\text{-}sibling}(x)
\end{aligned}$$

We conclude with two functions label : $Node \rightarrow \mathcal{L}_V$ and value : $Node \rightarrow String$ that return the label (name) and value of nodes respectively. We summarise their semantics as follows:

Node Kind	label	value
root	"#root"	children values
element	element name	children values
attribute	attribute name	attribute value (text())
text	"#text"	text value (text())

where "children values" is a string that results from the concatenation of the values of all children nodes, in document order (as relation $\ll$ instructs). All XML node functions are presented in Table 2.1.

$\mathsf{children} : Node \rightarrow Set(Node)$	
$\mathsf{children}(x)$	$= \{\, y \mid (x \rightarrow y) \wedge (\, \mathsf{isElement}(y) \vee \mathsf{isText}(y)\,)\,\}$
$\mathsf{parent} : Node \rightarrow Set_1(Node)$	
$\mathsf{parent}(x)$	$= \{\, y \mid (y \rightarrow x) \wedge (\, \mathsf{isElement}(y) \vee \mathsf{isRoot}(y)\,)\,\}$
$\mathsf{attributes} : Node \rightarrow Set(Node)$	
$\mathsf{attributes}(x)$	$= \{\, y \mid (x \rightarrow y) \wedge \mathsf{isAttribute}(y)\,\}$
$\mathsf{root} : Node \rightarrow Node$	
$\mathsf{root}(x)$	$= \{\, y \mid y \in \mathsf{parent}^*(x) \wedge \mathsf{isRoot}(y)\,\}$
$\mathsf{siblings} : Node \rightarrow Set(Node)$	
$\mathsf{siblings}(x)$	$= \{\, z \mid y \in \mathsf{parent}(x) \wedge z \in \mathsf{children}(y)\,\}$
$\mathsf{preceding} : Node \rightarrow Set(Node)$	
$\mathsf{preceding}(x)$	$= \{\, y \mid y \in \mathsf{children}^+(\mathsf{root}(x)) \wedge$ $y \notin \mathsf{parent}^+(x) \wedge y \ll x\,\}$
$\mathsf{following} : Node \rightarrow Set(Node)$	
$\mathsf{following}(x)$	$= \{\, y \mid y \in \mathsf{children}^+(\mathsf{root}(x)) \wedge$ $y \notin \mathsf{children}^+(x) \wedge x \ll y\,\}$
$\mathsf{preceding\text{-}sibling} : Node \rightarrow Set(Node)$	
$\mathsf{preceding\text{-}sibling}(x)$	$= \{\, y \mid y \in \mathsf{siblings}(x) \wedge y \ll x\,\}$
$\mathsf{following\text{-}sibling} : Node \rightarrow Set(Node)$	
$\mathsf{following\text{-}sibling}(x)$	$= \{\, y \mid y \in \mathsf{siblings}(x) \wedge x \ll y\,\}$

Table 2.1: Mathematical definitions of XML node relationships

2.1.1.1 Label Paths

We now turn our attention to label paths or simply *paths*. We first provide the definitions of *node paths* and *label paths*:

Definition 1.2: *The node path of a document tree node $x_n \in V$, is the node sequence $x_0 \ldots x_n$ where node $x_i \in parent^*(x_n)$ for $0 \leq i \leq n$ and $x_i \ll x_{i+1}$ for $0 \leq i \leq n-1$.* □

Definition 1.3: *The label path (path) of a node $x_n \in V$, is the label sequence $l_0/\ldots/l_n$, where each label l_i is produced by applying function* label *to each of the nodes of the node path $x_0 \ldots x_n$* i.e., *$l_i = \mathsf{label}(x_i)$ for $0 \leq i \leq n$.* □

The notion of a *path p* is of paramount importance for the definition of our storage model. To that end, we introduce the following terminology and functions regarding

$\mathsf{prefix} : Path \rightarrow Set(Path)$	
$\mathsf{prefix}(l_0/\ldots/l_n)$	$= \cup_{k=0}^{n} \{\ \ell^k(l_0/\ldots/l_n)\ \}$
$\mathsf{proper\_prefix} : Path \rightarrow Set(Path)$	
$\mathsf{proper\_prefix}(l_0/\ldots/l_n)$	$= \mathsf{prefix}(l_0/\ldots/l_{n-1})$
$\mathsf{maximal\_prefix} : Path \rightarrow Path$	
$\mathsf{maximal\_prefix}(l_0/\ldots/l_n)$	$= \ell^{n-1}(l_0/\ldots/l_n) = l_0/\ldots/l_{n-1}$
$\mathsf{isSuffix} : Path \times Path \rightarrow Boolean$	
$\mathsf{isSuffix}(l_0/\ldots/l_n, l'_0/\ldots/l'_k)$	$= k \leq n \wedge (l_i = l'_i, \text{for } i \in [0,k])$
$\mathsf{level} : Path \rightarrow Integer$	
$\mathsf{level}(l_0/\ldots/l_n)$	$= n-1$

Table 2.2: Path-related function definitions

paths. We begin by introducing type *Path*, as a synonym for "label sequence", a specialised *String* type that contains a sequence of labels (string literals over domain $\mathcal{L}_V$), separated by the distinguished character '/'. Function $\mathsf{path} : Node \rightarrow Path$ produces the *path* (*label-path*) $l_0/\ldots/l_n$ of a node as defined in Definition 1.3. Let us define an auxiliary function $\ell : Path \rightarrow Path$ as follows:

$$\ell^k(l_0/\ldots/l_n) = \begin{cases} l_0 & \text{if } k = 0 \\ \ell^{k-1}(l_0/\ldots/l_n)/l_k & \text{otherwise} \end{cases}$$

We can now further define functions $\mathsf{prefix}, \mathsf{proper\_prefix}, \mathsf{maximal\_prefix}, \mathsf{isSuffix}$ operating on a *Path* argument in a similar manner as their string function counterparts. In detail, function $\mathsf{prefix} : Path \rightarrow Set(Path)$ operates on a path $p = l_0/\ldots/l_n$ and computes (the set of) paths that are a prefix of p. Similarly, function $\mathsf{proper\_prefix} : Path \rightarrow Set(Path)$ computes (the set of) paths that are proper prefixes of path p (*i.e.,* excluding itself), while function $\mathsf{maximal\_prefix} : Path \rightarrow Path$ returns the maximal sub-path. Finally, function $\mathsf{isSuffix} : Path \times Path \rightarrow Boolean$ tests whether a path is a suffix of another path. We also define a function $\mathsf{level} : Path \rightarrow Integer$ that returns the level of all nodes with certain path value. Their definitions are presented in Table 2.2.

2.1.2 Location Paths

We now define a subset of the XPath 2.0 language [14] that is considered for the rest of this thesis. This subset contains only constructs that are useful or relevant to the query

reduction process. We denote this subset of XPath as $\mathcal{XP}$ and its (abstract) syntax is the following:

$$
\begin{aligned}
path &::= path \mid path \;\mid\; /\, path \;\mid\; path\,/\,path \;\mid\; path\,[\,qualif\,] \;\mid\; \\
&\quad axis :: nodetest \;\mid\; \varepsilon \\
qualif &::= qualif \text{ and } qualif \;\mid\; qualif \text{ or } qualif \;\mid\; \text{not } (qualif) \;\mid\; \\
&\quad (\,qualif\,) \;\mid\; path \;\mid\; path\ op\ value \\
axis &::= forward\ axis \;\mid\; reverse\ axis \\
forward\ axis &::= self \;\mid\; child \;\mid\; descendant \;\mid\; descendant\text{-}or\text{-}self \;\mid\; \\
&\quad following \;\mid\; following\text{-}sibling \\
reverse\ axis &::= parent \;\mid\; ancestor \;\mid\; ancestor\text{-}or\text{-self} \;\mid\; \\
&\quad preceding \;\mid\; preceding\text{-}sibling \\
op &::= \;<\; \mid\; \leq \;\mid\; >\; \mid\; \geq \;\mid\; =\; \mid\; \neq \\
nodetest &::= name \;\mid\; \star \;\mid\; node() \;\mid\; element() \;\mid\; text() \;\mid\; attribute()
\end{aligned}
$$

Central to $\mathcal{XP}$ (as in XPath in general) is the notion of a *Location Path Expression* or location path or *path* for short. A location path can be considered as the series of *location steps* taken to reach the node/nodes being selected. Note, that for the purpose of this chapter, we ignore any other notion of node ordering of the result of a location path expression apart from the document order. Location steps or steps for short, are path expressions of type: $step$ or $step[qualif]$ with $step$ being an $axis :: nodetest$ expression. A location step identifies a set of all nodes that are reachable from a context node with respect to the *axis* and *nodetest* specification. In the presence of a qualifier expression *qualif*, the set is filtered so that it only contains nodes that satisfy *qualif*. In addition, operator "|" results in the union of two path expressions while ε is the path expression that selects no node *i.e.,* returns an empty set.

A path expression of the form: $/path$ (*i.e.,* having a "/" at the beginning of the expression) is termed an "absolute path". Character "/" matches the document root node as it is defined in the specification of the XPath 2.0 Data Model [101]. The union of absolute paths also defines an absolute path. In any other case, a path expression is termed a "relative path".

2.1.3 Formal Semantics

We provide the denotational semantics of $\mathcal{XP}$, as specified by two functions $\mathcal{S}$, $\mathcal{Q}$. $\mathcal{S}[\![path]\!]x$ denotes the set of nodes selected by path expression *path* with node x being the context node. Similarly, $\mathcal{Q}[\![qualif]\!]x$ denotes whether qualifier *qualif* is satisfied for context node x.

Our definitions are largely adopted from [99, 100]. Again, we only focus on language $\mathcal{XP}$ and provide definitions that are useful in the scope of query reduction. In particular, the notion of order of the resulting node set of a path expression is ignored. Furthermore, we simplify the nodetest semantics, as it is not of great importance for the current work, by defining an auxiliary nodetest function τ. To that end, let *NodeTest* be the domain of node tests defined as $NodeTest = L_V \cup \{*, node(), element(), attribute(), text()\}$.

Function $\tau : Node \times NodeTest \rightarrow Boolean$ is defined as:

$$\tau(x,n) = \begin{cases} \mathsf{isElement}(x) \vee \mathsf{isAttribute}(x) & \text{if } n = * \\ \tau(x,*) \wedge n = \mathsf{label}(x) & \text{if } n \text{ is a name} \\ \textbf{true} & \text{if } n = node() \\ \mathsf{isElement}(x) & \text{if } n = element() \\ \mathsf{isAttribute}(x) & \text{if } n = attribute() \\ \mathsf{isText}(x) & \text{if } n = text() \end{cases}$$

The denotational semantics of $\mathcal{XP}$ is summarised in Table 2.3.

2.2 Striped Storage Model

It is well known that a key point in XML query processing is the efficient evaluation of structural relationships between XML elements, *i.e.,* structural joins (*e.g.,* [8, 18, 49]). In particular, ancestor-descendant and parent-child relationships play a central role due to the hierarchical model of XML. Many algorithms have been proposed in the literature and all operate on ordered lists of ancestor and descendant elements. An issue of great importance then is the efficient identification of such lists with respect to the query semantics.

Another crucial point in XML query evaluation is to minimise the input so that the query engine merely touches relevant data with respect to a given query. In [70],

$\mathcal{S} : Pattern \rightarrow Node \rightarrow Set(Node)$	
$\mathcal{S}[\![p_1 \mid p_2]\!]x$	$= \mathcal{S}[\![p_1]\!]x \cup \mathcal{S}[\![p_2]\!]x$
$\mathcal{S}[\![/p]\!]x$	$= \mathcal{S}[\![p]\!](\mathsf{root}(x))$
$\mathcal{S}[\![p_1/p_2]\!]x$	$= \{ x_2 \mid x_1 \in \mathcal{S}[\![p_1]\!]x \wedge x_2 \in \mathcal{S}[\![p_1]\!]x_1 \}$
$\mathcal{S}[\![p[q]]\!]x$	$= \{ x_1 \mid x_1 \in \mathcal{S}[\![p]\!]x \wedge \mathcal{Q}[\![q]\!]x_1 \}$
$\mathcal{S}[\![(p)]\!]x$	$= \mathcal{S}[\![p]\!]x$
$\mathcal{S}[\![\varepsilon]\!]x$	$= \emptyset$
$\mathcal{S}[\![self :: n]\!]x$	$= \{ x_1 \mid x_1 = x \wedge \tau(x_1, n) \}$
$\mathcal{S}[\![attribute :: n]\!]x$	$= \{ x_1 \mid x_1 \in \mathsf{attributes}(x) \wedge \tau(x_1, n) \}$
$\mathcal{S}[\![child :: n]\!]x$	$= \{ x_1 \mid x_1 \in \mathsf{children}(x) \wedge \tau(x_1, n) \}$
$\mathcal{S}[\![descendant :: n]\!]x$	$= \{ x_1 \mid x_1 \in \mathsf{children}^+(x) \wedge \tau(x_1, n) \}$
$\mathcal{S}[\![descendant\text{-}or\text{-}self :: n]\!]x$	$= \{ x_1 \mid x_1 \in \mathsf{children}^*(x) \wedge \tau(x_1, n) \}$
$\mathcal{S}[\![following :: n]\!]x$	$= \{ x_1 \mid x_1 \in \mathsf{following}(x) \wedge \tau(x_1, n) \}$
$\mathcal{S}[\![following\text{-}sibling :: n]\!]x$	$= \{ x_1 \mid x_1 \in \mathsf{following\text{-}sibling}(x) \wedge \tau(x_1, n) \}$
$\mathcal{S}[\![parent :: n]\!]x$	$= \{ x_1 \mid x_1 \in \mathsf{parent}(x) \wedge \tau(x_1, n) \}$
$\mathcal{S}[\![ancestor :: n]\!]x$	$= \{ x_1 \mid x_1 \in \mathsf{parent}^+(x) \wedge \tau(x_1, n) \}$
$\mathcal{S}[\![ancestor\text{-}or\text{-}\mathrm{self} :: n]\!]x$	$= \{ x_1 \mid x_1 \in \mathsf{parent}^*(x) \wedge \tau(x_1, n) \}$
$\mathcal{S}[\![preceding :: n]\!]x$	$= \{ x_1 \mid x_1 \in \mathsf{preceding}(x) \wedge \tau(x_1, n) \}$
$\mathcal{S}[\![preceding\text{-}sibling :: n]\!]x$	$= \{ x_1 \mid x_1 \in \mathsf{preceding\text{-}sibling}(x) \wedge \tau(x_1, n) \}$
$\mathcal{Q} : Qualifier \rightarrow Node \rightarrow Boolean$	
$\mathcal{Q}[\![q_1 \,\mathrm{and}\, q_2]\!]x$	$= \mathcal{Q}[\![q_1]\!]x \wedge \mathcal{Q}[\![q_2]\!]x$
$\mathcal{Q}[\![q_1 \,\mathrm{or}\, q_2]\!]x$	$= \mathcal{Q}[\![q_1]\!]x \vee \mathcal{Q}[\![q_2]\!]x$
$\mathcal{Q}[\![\mathrm{not}(q)]\!]x$	$= \neg \mathcal{Q}[\![q]\!]x$
$\mathcal{Q}[\![p \; op \; val]\!]x$	$= \{ x_1 \mid x_1 \in \mathcal{S}[\![p]\!]x \wedge (\mathsf{value}(x_1) \; op \; val) \} \neq \emptyset$
$\mathcal{Q}[\![p]\!]x$	$= \mathcal{S}[\![p]\!]x \neq \emptyset$

Table 2.3: Denotational semantics of $\mathcal{XP}$

the authors introduced the notion of XML projection and their study concluded that in most cases only a small percentage of the original XML document is needed in order to evaluate a given query.

Motivated by the above processing paradigms, we focus on a storage model that efficiently provides structural join operators with appropriate input lists and at the same time strives for input list minimisation with respect to the query and the input XML document. We treat XML containment relationships (mainly ancestor-descendant and parent-child relationships) as first-class citizens due to their significance in XML query evaluation. As such, we aim for the efficient identification of all nodes ($n \in T$) satisfying a simple path expression or path p.

To that end, we employ a vertical partitioning decomposition method, which we term *Striping*. Instead of clustering XML tree nodes along with their children, we partition them according to their (label-)path p. We term these partitions *Stripes*. Borrowing ideas from XMill, an XML semantic compressor [64], we separate the tree structure of the document from the actual data values, resulting in three distinct types of Stripes: *path*, *attribute* and *value* Stripes. Each Stripe is labelled with the path p it represents. There exists a single *Path/Attribute/Value Stripe* S_p, for each unique path p containing all element/attribute/text nodes n respectively, that satisfy predicate $\mathsf{path}(n) = p$. We now define our *striped model* as follows:

Definition 2.1: *The striped representation of an* XML *document tree* T *is the triplet* $\mathcal{SRX}(T) = (\mathcal{P}, \mathcal{A}, \mathcal{V})$ *where:*

- $\mathcal{P}$ *is the set of Stripes that contain all element nodes of* T *(path Stripes);*
- $\mathcal{A}$ *is the set of Stripes that contain all attribute nodes of* T *(attribute Stripes);*
- $\mathcal{V}$ *is the set of Stripes that contain all data nodes of* T *(value Stripes).*

□

A Stripe can be considered as a conceptual grouping of XML nodes that share certain properties regardless of their type. To focus on these properties, we define the *Abstract Stripe* or simply *Stripe*, and describe its main properties:

Definition 2.2: *A Stripe* S_p *of a valid path* p *in document* T*, is the set of all* XML *nodes* $u \in V_T$ *that belong to path* p*. Formally:* $S_p = \{\, u \,|\, u \in V_T \wedge p = \mathsf{path}(u) \,\}$. □

Proposition 2.1: *All tree nodes $u \in V_T$ of a document T that are members of Stripe S_p, share the same path which is the Stripe path* i.e., $u \in S_p \Leftrightarrow \mathsf{path}(u) = p$. □

Proof: Directly from Definition 2.2. □

From Proposition 2.1, we have that since all nodes that are members of a Stripe S_p share the same path value, they also share the same label and level value. We exploit that observation and overload functions label, level, path to operate on Stripes in addition to document nodes and return the value of label, level and path of all nodes of a Stripe respectively. A description of all Stripe-related functions is shown in Table 2.4.

Example 2.1: Consider Stripe $S_{bib/book/author}$ of the striped representation of the document tree depicted in Figure 2.1. If we apply functions label, level and path for Stripe $S_{bib/book/author}$ or on any of its nodes, we obtain the same results. We also demonstrate the close relationship of a Stripe with its path:

- $\mathsf{label}(S_{bib/book/author}) = author$
- $\mathsf{level}(S_{bib/book/author}) = 2$
- $\mathsf{path}(S_{bib/book/author}) = bib/book/author$
- $\mathsf{proper\_prefix}(S_{bib/book/author}) = \{S_{bib},\ S_{bib/book}\}$
- $\mathsf{prefix}(\mathsf{path}(S_{bib/book/author})) = \{bib,\ bib/book,\ bib/book/author\}$

Note, that we ignore the label of the document root node "#root" as it is common for all Stripes. □

Proposition 2.2: *For any tree node $u \in V_T$ of a document T that is a member of Stripe S_p, there exists at least one ancestor tree node in each Stripe $S_{p'}$ where $p' \in \mathsf{proper\_prefix}(p)$. In detail:*

1. *For Stripe $S_{p'}$ with $p' = \mathsf{maximal\_prefix}(p)$, there exists a node $u' \in S_{p'}$ such that $u' \in \mathsf{parent}(u)$.*
2. *For any Stripe $S_{p'}$ with $p' \in \mathsf{proper\_prefix}(p) - \{\ \mathsf{maximal\_prefix}(p)\ \}$, there exists a node $u' \in S_{p'}$ such that $u' \in \mathsf{parent}^+(u)$.*

□

Proof: Since node u is a member of Stripe S_p, from Proposition 2.1, we have that $\mathsf{path}(u) = p$. Let p be the label sequence $l_0/\ldots/l_n$, which also implies that $\mathsf{label}(u) =$

Function		Description
label	: $Stripe \rightarrow \mathcal{L}_V$	returns Stripe label
level	: $Stripe \rightarrow Integer$	returns Stripe level
path	: $Stripe \rightarrow Path$	returns Stripe label-path
prefix	: $Stripe \rightarrow Set(Stripe)$	returns the set of all rooted sub-paths, including itself
proper_prefix	: $Stripe \rightarrow Set(Stripe)$	returns the set of all rooted sub-paths (excludes itself)
maximal_prefix	: $Stripe \rightarrow Stripe$	returns its rooted maximal sub-path

Table 2.4: Stripe-related functions

l_n. Now, the existence of a node u in path $l_0/\ldots/l_n$ implies that there exists a node path (*i.e.,* a node sequence) $u_0 \ldots u_n$ such that: *a*) node u_o matches the document root r; *b*) node u_n matches node u and *c*) each of the nodes in sequence $u_0 \ldots u_{n-1}$ are ancestor nodes of u with node u_{n-1} in particular, to being the parent node of u_n. In addition, the striped model ensures that all nodes in sequence $u_0 \ldots u_{n-1}$ will be members of Stripes over paths $\mathsf{path}(u_0), \mathsf{path}(u_1), \ldots, \mathsf{path}(u_{n-1})$ respectively, which are all paths $p' \in \mathsf{proper\_prefix}(l_0/\ldots/l_n)$. □

2.3 Stripe Processing

In relational database systems, the SQL compiler is responsible for parsing a query and identifying all relations and their corresponding attributes that contain all relevant to the query data. In our striped representation, we use Stripes as the building block to store XML data, in analogy to relations that are used for storing data in a relational database system. However, Stripes maintain structural information about data they enclose with the path (label-path) being the most significant. Our focus in this section is to determine any opportunities presented by the striped representation of an hierarchical storage model and describe how a query compiler can exploit them to minimise query input size. We term this process *Input Minimisation* and divide it into two parts:

Stripe Projection given a query $q \in \mathcal{XP}$, project an initial set of Stripes that corresponds to the query input.

Stripe Pruning prune the initial set of Stripes (produced by stripe projection) accord-

ing to the (overall) query semantics.

We now present some definitions that are used extensively in the rest of the chapter. We have already introduced the type of a Stripe (Section 2.2), as a set of XML Nodes *i.e.,* $Set(Node)$. We now introduce the type of a set of Stripes: $Set(Stripe)$. We use the terms *NodeSet* and *StripeSet* as shorthand for $Set(Node)$ and $Set(Stripe)$ types respectively. A StripeSet $\mathcal{SS}$ of type *StripeSet* is a set of elements of type *Stripe*, which in turn are sets of elements of type *Node*. We define function $\mathsf{ss\_nodes} : Set(Stripe) \rightarrow Set(Node)$ that takes a StripeSet argument $\mathcal{SS}$ and results in a NodeSet that contains all nodes that are contained in all Stripes $S \in \mathcal{SS}$ *i.e.,*

$$\mathsf{ss\_nodes}(\mathcal{SS}) = \{\, x \,|\, S \in \mathcal{SS} \wedge x \in S \,\}$$

2.3.1 Stripe Dependency Graph

The semantics of $\mathcal{XP}$ (and XPath in general) specifies that for the evaluation of a path expression of the form: $path_1 \,/\, path_2$, the resulting (output) set of nodes from evaluating sub-expression $path_1$ is used as the set of context nodes (input) for evaluating sub-expression $path_2$. The existence of predicates and binary relationships between path expressions in $\mathcal{XP}$ though, results in situations where a set of nodes (*i.e.,* a *NodeSet*) can become the input NodeSet for multiple expressions or have multiple NodeSets as input. In order to capture such dependencies and considering the location step $a_i{::}n_i$ as the core expression of any $\mathcal{XP}$ expression, we construct a graph structure, the *Stripe Dependency Graph* or *Stripe Graph* for short.

A Stripe Graph of a query q is a directed, acyclic graph (DAG) $G = (V, E)$, where V is the vertex set which corresponds to the set of query terms while E is the edge set which defines their relationships. For simplicity, we consider a single dependency graph for each absolute path expression, *i.e.,* the root node of the graph corresponds to the document root. We logically divide the vertex set V in two disjoint sets of graph vertices V_s and V_b. Each time a location step expression $a_i{::}n_i$ is encountered, we create a *step* vertex $v \in V_s$ annotated with nodetest n_i and add an edge $(u \xrightarrow{a_i} v)$ for every vertex $u \in V$ that corresponds to an expression which produces input for n_i. In general, an edge $(u \xrightarrow{a_i} v)$ annotated with an axis enforces the axis relationship between graph vertices. In the presence of binary operators, we add a special *binary* vertex $u \in V_b$, annotated with the operation type (conjunction or disjunction), to capture the correct operator precedence order and semantics.

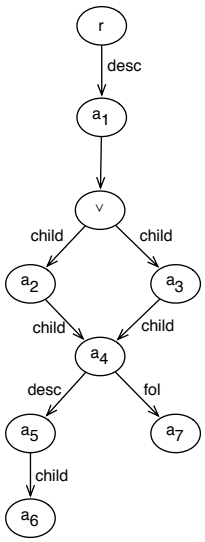


Figure 2.2: Stripe Graph for query q_1 : *descendant*::$a_1/(a_2|a_3)/a_4[$*descendant*::$a_5/a_6]/$ *following*::a_7

Example 2.2: Consider query q_1 :

$$descendant::a_1/(a_2\,|\,a_3)/a_4[descendant::a_5/a_6]/following::a_7$$

We traverse query q_1 (its syntax tree to be precise) and construct its Stripe Graph, shown in Figure 2.2: We first add the graph root r for document root. Then we add step vertex a_1 and edge $(r \xrightarrow{desc} a_1)$ which is interpreted as "from document root, we project all a_1 descendant nodes". For the union of path expressions: a_2, a_3, we add a special disjunction vertex *disj* (labelled as "$\vee$" at the Stripe Graph), reflecting the union operator semantics, and connect that to step vertex a_1 as it will provide the input for both subexpressions. We then handle the a_2 and a_3 subexpressions by creating two vertices, one for each expression, and we add edges $(disj \xrightarrow{child} a_2)$ and $(disj \xrightarrow{child} a_3)$ as both sub-expressions have a_1 nodes as their context nodes. Likewise, we connect vertex a_4 with both vertices a_2 and a_3 for the next step. We continue by adding edges $(a_4 \xrightarrow{desc} a_5)$ and $(a_5 \xrightarrow{child} a_6)$ for the predicate expression and finally, edge $(a_4 \xrightarrow{fol} a_7)$. Note here that the two edges originating from vertex a_4 have conjunction semantics, although not explicitly stated, as they enforce both axis relationships between vertex a_4 and vertices a_6, a_7 respectively. □

Each vertex $u \in V$ of the Stripe Graph of a query $q \in \mathcal{XP}$ is assigned an (initially empty) StripeSet $u.\mathcal{SS}$. This StripeSet will be later populated with the Stripes that contain the candidate nodes for the evaluation of the expression that corresponds to vertex u. In particular, in the case of a *step* vertex $u \in V_s$, StripeSet $u.\mathcal{SS}$ will contain

Stripes that contain all possible candidate nodes regarding location step $a_i::n_i$, where n_i is the nodetest annotation of vertex u and a_i is the axis annotation of an incoming edge. We now define the *Query Input* and *Result StripeSet* as follows:

Definition 3.1: *The Query Input I_q of a query $q \in \mathcal{XP}$ is the set of all StipeSets for all step vertices of the Stripe Graph for query q. More formally: $I_q = \{ u.\mathcal{SS} \,|\, u \in V_s \}$, where V_s is the set of step vertices of q's Stripe Graph.* □

Definition 3.2: *The Result StripeSet $\mathcal{RS}_q$ of a query $q \in \mathcal{XP}$ is the union of all StripeSets of the Stripe Graph vertices that correspond to the final result of the query (output nodes).* □

2.3.2 Stripe Projection

Stripe projection for a given query $q \in \mathcal{XP}$ is the process of identifying the set of StripeSets I_q that contain all nodes necessary to evaluate query q over an XML tree T. We use the term *projection* as for a given query q it projects out StripeSets from StripeSet $\mathcal{SS}_T$, where $\mathcal{SS}_T$ is the StripeSet containing all (path, attribute and value) Stripes created for document tree T *i.e.,* $\mathcal{SS}_T = \mathcal{P}_T \cup \mathcal{A}_T \cup \mathcal{V}_T$, as defined in Definition 2.1.

Let us for now consider that query q is a predicate-free step expression *i.e.,* a sequence of $\mathcal{XP}$ location steps with no predicate (filter) expressions: $/a_1::n_1/a_2::n_2/\ldots/a_k::n_k$, where n_i is a nodetest while a_i is any of the $\mathcal{XP}$ axes as described in Section 2.1.2 for any $i \in [1,k]$. The Stripe Graph for such expressions would look like:

$$r \xrightarrow{a_1} n_1 \xrightarrow{a_2} n_2 \ldots \xrightarrow{a_k} n_k$$

Starting from the leftmost edge of q's Stripe Graph, Stripe projection operates on each incoming vertex u_i in two phases:

Axis Projection First a candidate StripeSet $u_i.\mathcal{SS}$ is projected from StripeSet $\mathcal{SS}_T$. This is the set of Stripes where all candidate nodes for the evaluation of step $a_i :: *$ reside. The projection occurs according to the step axis semantics of a_i and having as input set $u_{i-1}.\mathcal{SS}$, the StripeSet previously projected for vertex u_{i-1}, which contains all Stripes where all context nodes for step i reside. We begin with $r.\mathcal{SS} = \mathcal{SS}_T$.

Nodetest Filtering Candidate StripeSet $u_i.\mathcal{SS}$ is then filtered according to nodetest

semantics of n_i to formulate the final set for vertex u_i. It then becomes the context StripeSet for vertex u_{i+1}.

The process continues until vertex u_k is processed. The query input and result StripeSet for query q are:

$$I_q = \{ u_i.\mathcal{SS} \mid 1 \leq i \leq k \}$$
$$\mathcal{RS}_q = u_k.\mathcal{SS}$$

The projection process effectively means that for each vertex $u_i \in V$ of the Stripe Graph, StripeSet $u_i.\mathcal{SS}$ defines the domain for the evaluation of the corresponding location step $a_i::n_i$ *i.e.,* for any location step $a_i::n_i$ (u_i) and a context node x the following holds:

$$\mathcal{S}[\![a_i::n_i]\!]x \subseteq \mathsf{ss\_nodes}(u_i.\mathcal{SS}) \tag{2.1}$$

while in particular for the *Result StripeSet* of q:

$$\mathcal{S}[\![q]\!] \subseteq \mathsf{ss\_nodes}(\mathcal{RS}_q) \tag{2.2}$$

We now describe the two phases of Stripe projection operating on a single Stripe Graph edge $(u \xrightarrow{a} v)$.

2.3.2.1 Axis Projection

This phase can be considered as StripeSet projection without considering nodetest $v.n$. This is the most general case since no filtering applies. The objective in this phase is to project StripeSet $v.\mathcal{SS}$ from StripeSet $\mathcal{SS}_T$ with respect to axis a and a given reference StripeSet $u.\mathcal{SS}$ that contains all context nodes. Stripe projection proceeds as follows:

- *Self* axis: All axis *self* candidate nodes will still reside in the same path(s) as the context nodes. Therefore, we have that:

$$v.\mathcal{SS} = \{ S_{p'} \mid S_p \in u.\mathcal{SS} \wedge S_{p'} \in \mathcal{SS}_T \wedge p = p' \} = u.\mathcal{SS}$$

- *Child* axis: We need to project the set of all Stripes of $\mathcal{SS}_T$ that contain the *children* nodes of the context nodes included in Stripes of StripeSet $u.\mathcal{SS}$. Therefore, we have that:

$$v.\mathcal{SS} = \{ S_{p'} \mid S_p \in u.\mathcal{SS} \wedge S_{p'} \in \mathcal{SS}_T \wedge p = \mathsf{maximal\_prefix}(p') \}$$

- *Attribute* axis: We identify $u.\mathcal{SS}$ in the same manner as with the *child* axis but use attribute Stripes:

$$v.\mathcal{SS} = \{ S_{p'} \,|\, S_p \in u.\mathcal{SS} \wedge S_{p'} \in \mathcal{A}_T \wedge p = \mathsf{maximal\_prefix}(p') \}$$

- *Descendant* axis: Since it is defined as the transitive closure of the *child* axis [14], the same logic is applied for StripeSet identification:

$$v.\mathcal{SS} = \{ S_{p'} \,|\, S_p \in u.\mathcal{SS} \wedge S_{p'} \in \mathcal{SS}_T \wedge p \in \mathsf{proper\_prefix}(p') \}$$

- *Descendant-or-self* axis: As the axis name instructs, the candidate StripeSet is the union of the StripeSet projected *w.r.t.* the *descendant* axis (denoted as $\mathcal{SS}^{desc}$) and the StripeSet projected *w.r.t.* the *self* axis (denoted as $\mathcal{SS}^{self}$). Indeed we have:

$$\begin{aligned} v.\mathcal{SS} &= \mathcal{SS}^{desc} \cup \mathcal{SS}^{self} \\ &= \{ S_{p'} \,|\, S_p \in u.\mathcal{SS} \wedge S_{p'} \in \mathcal{SS}_T \wedge (p \in \mathsf{proper\_prefix}(p') \vee p = p') \} \\ &= \{ S_{p'} \,|\, S_p \in u.\mathcal{SS} \wedge S_{p'} \in \mathcal{SS}_T \wedge p \in \mathsf{prefix}(p') \} \end{aligned}$$

- *Following* axis: In this case, there is no containment relationship between a context node and its candidate nodes *w.r.t.* the *following* axis. A context node, say with path p, may have a following node in any path p' of document T, including path p itself. Therefore, we cannot exclude any Stripe from $\mathcal{SS}_T$ *i.e.,* $v.\mathcal{SS} = \mathcal{SS}_T$.

- *Following-sibling* axis: As in the case of the *following* axis, there is no containment property between a context node and its candidate nodes *w.r.t.* the *following-sibling* axis. Nevertheless, we exploit their sibling relationship and therefore project the candidate StripeSet as follows:

$$v.\mathcal{SS} = \{ S_{p'} \,|\, S_p \in u.\mathcal{SS} \wedge S_{p'} \in \mathcal{SS}_T \wedge \mathsf{maximal\_prefix}(p') = \mathsf{maximal\_prefix}(p) \}$$

For any of the reverse axes, Stripe projection occurs in a similar manner as at the projection of its "symmetrical" (as specified in [81]) axis. Table 2.5 summarises the projection of StripeSet $v.\mathcal{SS}$ from $\mathcal{SS}_T$ given StripeSet $u.\mathcal{SS}$ and for any of the reverse axes:

Reverse Axis	$v.\mathcal{SS}$
parent	$\{ S_{p'} \mid S_p \in u.\mathcal{SS} \wedge S_{p'} \in \mathcal{SS}_T \wedge p' = \mathsf{maximal\_prefix}(p) \}$
ancestor	$\{ S_{p'} \mid S_p \in u.\mathcal{SS} \wedge S_{p'} \in \mathcal{SS}_T \wedge p' \in \mathsf{proper\_prefix}(p) \}$
ancestor-or-self	$\{ S_{p'} \mid S_p \in u.\mathcal{SS} \wedge S_{p'} \in \mathcal{SS}_T \wedge p' \in \mathsf{prefix}(p) \}$
preceding	$\mathcal{SS}_T$
preceding-sibling	$\{ S_{p'} \mid S_p \in u.\mathcal{SS} \wedge S_{p'} \in \mathcal{SS}_T \wedge \mathsf{maximal\_prefix}(p') = \mathsf{maximal\_prefix}(p) \}$

Table 2.5: StripeSet projection based on reverse axes

2.3.2.2 Nodetest Filtering

Nodetest filtering is the process where the projected StripeSet is filtered against nodetest n. To that end, we define a function $\mathsf{filter\_nodetest} : Stripe \times NodeTest \rightarrow Boolean$ as follows:

$$\mathsf{filter\_nodetest}(S_p, n) = \begin{cases} S_p \in \mathcal{P}_T \cup \mathcal{A}_T & \text{if } n = * \\ \mathsf{filter\_nodetest}(S_p, *) \wedge \mathsf{isSuffix}(S_p, n) & \text{if } n \text{ is a name} \\ \textbf{true} & \text{if } n = node() \\ S_p \in \mathcal{P}_T & \text{if } n = element() \\ S_p \in \mathcal{A}_T & \text{if } n = attribute() \\ S_p \in \mathcal{V}_T & \text{if } n = text() \end{cases}$$

When nodetest n is a nametest (a name or a wildcard $*$), all value Stripes need to be filtered out. In the case of a wildcard, no filtering is necessary since it satisfies all selected Stripes from the axis projection phase. If the nametest is a specific element/attribute name then the filtering process will discard all Stripes that (a) are not of the requested type, and (b) do not satisfy predicate suffix; it is certain that these Stripes do not contain any of the requested nodes. When nodetest n is a kindtest, then it drives the projection process according to the node kind: Tests $element(), attribute(), text()$ process only Path, Attribute or Value Stripes respectively while test $node()$ processes any type of Stripes.

As a final note, nodetest filtering does not have to run on top of axis projection as a separate process. It is rather naturally applied on-the-fly during the axis-projection of candidate Stripes. Consider, *e.g.,* location step *child*::*name* that would project candi-

date StripeSet (given input context StripeSet $\mathcal{SS}_{ctx}$) as:

$$\mathcal{SS} = \{\, S_{p'} \,|\, S_p \in \mathcal{SS}_{ctx} \wedge S_{p'} \in \mathcal{P}_T \wedge p = \mathsf{maximal\_prefix}(p') \wedge \mathsf{isSuffix}(p', name) \,\}$$

Node that the candidate Stripes are now selected directly from $\mathcal{P}_T$ since the semantics of the location step instruct the selection of element nodes.

2.3.2.3 Content-Aware Stripe Projection

We also describe a special case of Stripe projection which is based on the semantics of value-based predicate expressions. The basic Stripe projection process computes an initial StripeSet I_q from StripeSet $\mathcal{SS}_T$ that corresponds to the input query for given query $q \in \mathcal{XP}$. However, the projection process mainly exploits navigational parts of query q. In addition to the navigational information we can extract from a given query, we can further eliminate input stripes based on value filtering. For instance, consider query $child::a[author/value() = \text{“}Smith''']$, having a single value predicate. When projecting the StripeSet for the graph vertex that corresponds to the kindtest expression $value()$, we can take advantage of the value predicate and test whether candidate Stripes contain nodes satisfying the predicate. For this purpose, we keep minimal value statistics for the Attribute and Value Stripes at the Stripe signatures (metadata). If a Stripe does not contain any node satisfying the predicate then it is filtered out from the projection StripeSet.

As a final note, for enabling content-aware Stripe projection, we need to enhance the Stripe Graph by annotating the corresponding vertices with the predicate attributes (condition and test value). This process can then be incorporated in the basic projection process.

2.3.2.4 Projection algorithm

We now present the algorithm for Stripe projection over the Stripe Graph G of query q (shown in Figure 2.3). The traversal of a graph edge $(u \xrightarrow{a} v)$ (implying that $v \in V_s$) instructs the projection of a StripeSet from $\mathcal{SS}_T$, having as input StripeSet $u.\mathcal{SS}$ and filtered with $v.n$ nodetest. In addition, the traversal of an edge $(u \rightarrow v)$ with vertex v being a special disjunction or conjunction vertex, simply copies the StripeSet of the originating vertex to the destination vertex. We delegate this process to function project_stripes which operates as described in Sections 2.3.2.1, 2.3.2.2. The main idea of the algorithm is to visit each vertex u exactly once for each of its incoming edges in

order to project the final StripeSet $u.\mathcal{SS}$ having considered all input (StripeSets). We perform a depth-first traversal of the graph and for each edge $(u \xrightarrow{a} v) \in E$, we project all relevant Stripes (function project_stripes in line 2). Then we mark that vertex v has been visited through vertex u (function visited, line 3) and we recursively proceed with v's outgoing edges (lines 5 - 6) but only after v has been visited through all its incoming edges (function exhausted, line 4). This ensures that if a vertex u has multiple incoming edges, we will first project *all* relevant Stripes for vertex u before continuing to any other vertices that depends on u. The final StripeSet $u.\mathcal{SS}$ for a vertex u is the union of the results of project_stripes (line 2).

Example 2.3: Figure 2.4 illustrates a Stripe projection running example for query q_1. The process starts with edge $(r \xrightarrow{desc} a_1)$, shown in bold in Figure 2.4(a). After Stripe projection for vertex a_1, we mark the processed edge as visited. Vertex a_1 is now exhausted (drawn as a double-lined circle), and therefore we continue by traversing its outgoing edges. Likewise, disjunction vertex *disj* is visited through edge $(a_1 \rightarrow disj)$ (Figure 2.4(b)) and its StripeSet is set equal to that of vertex a_1, *i.e.,* $disj.\mathcal{SS} = a_1.\mathcal{SS}$. The process continues in the same manner until we reach vertex a_4 from a_2 as displayed in Figure 2.4(d). After the projection takes place, the process will not proceed to any of its outgoing vertices since it has not yet been visited from all its incoming edges. Therefore, the process "backtracks" and continues with edge $(disj \xrightarrow{child} a_3)$ (Figure 2.4(e)). When vertex a_4 is eventually processed from its second incoming edge (Figure 2.4(f)), and thus all relevant Stripes according to both inputs have been projected, the process will continue with its outgoing edges (Figures 2.4(g) - (i)). The process terminates when all vertices have been visited through all of their inputs and each vertex holds the projected StripeSet. According to our definitions, we have that $I_{q_1} = \{\, a_i.\mathcal{SS} \mid 1 \leq i \leq 7 \,\}$ while $\mathcal{RS}_{q_1} = a_7.\mathcal{SS}$. Note that the projected StripeSet of the disjunction vertex *disj* is not included in I_{q_1}. □

Note that Algorithm project() visits and also processes (*i.e.,* projects Stripes) each vertex as many times as the number of its incoming edges. Thus, function project_stripes is invoked $|E|$ times in total for Stripe projection. Algorithm project() performs a full traversal of the query Stripe Graph and therefore takes $O(|V| + |E|)$ time in order to initialise and visit all vertices. Let t_{prj} be the running time of function project_stripes, we then have that the running time of Algorithm project() is $O(|V| + |E| t_{prj})$.

Algorithm 2.1: project($Edge : e$)

Data: $e : u \xrightarrow{a} v$

Result: Stripe projection of $v.\mathcal{SS}$

```
1 begin
2     v.SS ← v.SS ∪ project_stripes(u.SS, v.n, a);
3     visited(v, u);
4     if (exhausted(v)) then
5         forall e' ∈ out_edges(v) do
6             project(e')
7 end
```

Figure 2.3: Stripe Projection algorithm

2.3.3 Stripe Pruning

We described the Stripe projection process which determines the relevant Stripes for correctly answering a query $q \in \mathcal{XP}$. A natural question that arises is whether this collection of Stripes, as projected for each location step of a query q, is the minimum set of Stripes needed to evaluate query q. The answer is that, no, in many cases it is not. The reason is that during Stripe projection, we merely project candidate StripeSets according to the input StripeSets and *w.r.t.* a navigation axis, without any consideration of the overall query semantics.

To that end, we employ a *Stripe Pruning* process, which, as its name suggests, prunes the projected StripeSets in I_q for query q according to q's semantics. Stripe pruning is the second step of the *Input Minimisation* process. In the same spirit as Stripe projection, Stripe pruning recursively visits each of the vertices of the Stripe Graph. For each visiting vertex u, it prunes StripeSet $u.\mathcal{SS}$ with respect to their adjacent vertices (StripeSets) and edge axis annotation a. Let us define function prune_stripes : $StripeSet \times StripeSet \times Axis \rightarrow StripeSet$ that prunes a StripeSet (first argument) with respect to another (second argument) and the axis relationship (third argument) and returns the resulting StripeSet. For instance, prune_stripes$(u.\mathcal{SS}, v.\mathcal{SS}, a)$ will return the result of pruning $u.\mathcal{SS}$ *w.r.t.* $v.\mathcal{SS}$ and axis a. In general, pruning occurs according to the axis projection rules defined in Section 2.3.2.1. For instance if a is the *child* axis in the above example, then function prune_stripes will return a StripeSet containing those Stripes from $u.\mathcal{SS}$ that their path p is a maximal_prefix of any Stripe at StripeSet $v.\mathcal{SS}$.

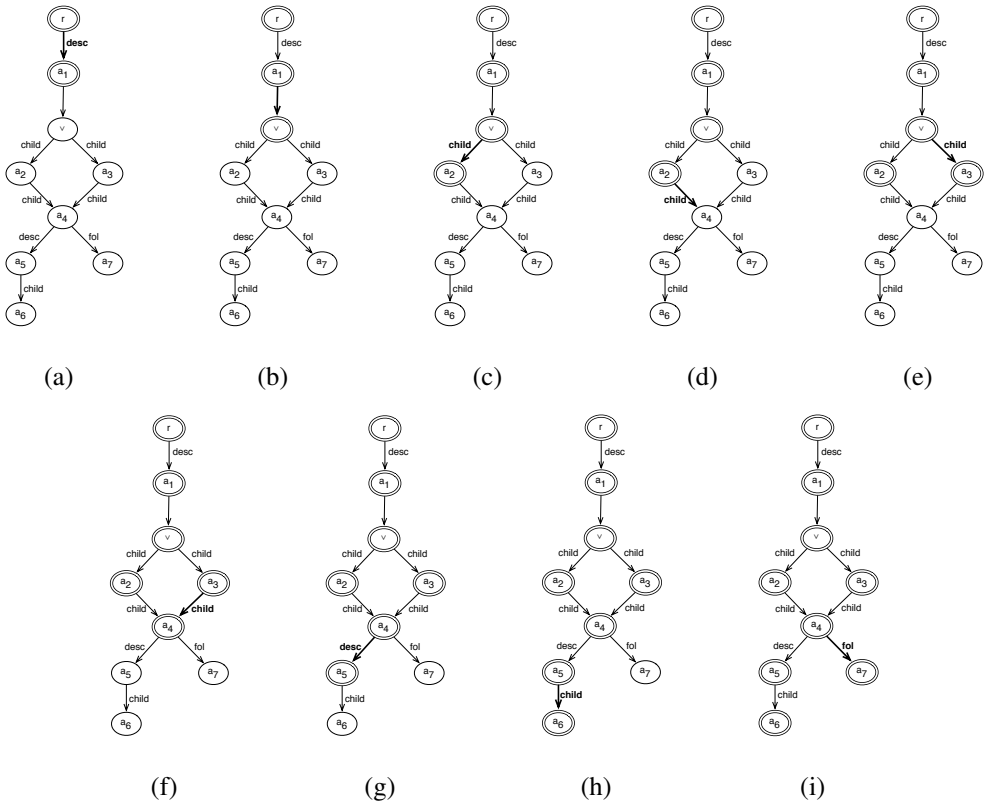


Figure 2.4: Stripe Projection running example for query q_1 : *descendant*::$a_1/(a_2|a_3)/a_4[$*descendant*::$a_5/a_6]/$*following*::a_7

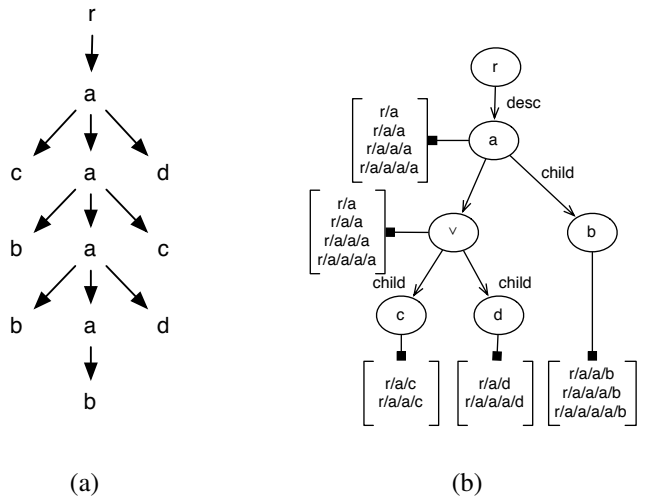
In addition, if the edge axis annotation is null *i.e.,* the destination vertex is a binary operator vertex, then $\mathsf{prune\_stripes}(u.\mathcal{SS}, v.\mathcal{SS}, \perp)$ simply returns StripeSet $v.\mathcal{SS}$.

We now define the following pruning rules in order to satisfy overall query semantics and cover $\mathcal{XP}$ binary operators:

1. When vertex u has a single outgoing edge $u \xrightarrow{a} v$ then we can directly prune u's StripeSet *w.r.t.* v: $u.\mathcal{SS} = \mathsf{prune\_stripes}(u.\mathcal{SS}, v.\mathcal{SS}, a)$.

2. When vertex u has multiple outgoing edges $u \xrightarrow{a_i} v_i$ and it is **not** a disjunction vertex, then we compute the final StripeSet as the intersection of pruning in isolation u's StripeSet *w.r.t.* each of its adjacent vertices v_i: $u.\mathcal{SS} = \bigcap_{(u \xrightarrow{a_i} v_i)} \mathsf{prune\_stripes}(u.\mathcal{SS}, v_i.\mathcal{SS}, a_i)$.

3. When vertex u has multiple outgoing edges $u \xrightarrow{a_i} v_i$ and it is a disjunction vertex, then we compute the final StripeSet as the union of pruning in isolation u's StripeSet *w.r.t.* each of its adjacent vertices v_i: we compute $u.\mathcal{SS}$ as: $u.\mathcal{SS} = \bigcup_{(u \xrightarrow{a_i} v_i)} \mathsf{prune\_stripes}(u.\mathcal{SS}, v_i.\mathcal{SS}, a_i)$.



(a) (b)

Figure 2.5: (a) A sample document tree and (b) Stripe Graph G for query q_2 : *descendant*::$a[c$ **or** $d]/b$

Example 2.4: Consider query q_2 : *descendant*::$a[c$ **or** $d]/b$ against the document tree T shown in Figure 2.5(a). Query q_2 returns a node sequence of b nodes that their a parent nodes have *at least one* c child node or d child node. Figure 2.5(b) depicts the Stripe dependency graph for query q_2 and the projected StripeSets for each vertex as produced by Stripe projection. Now, let us consider pruning the StripeSet of vertex a with respect to its adjacent vertices. In order to do that, we first need to prune vertex $disj$'s StripeSet with respect to its adjacent vertices c and d. By applying pruning rule 3, we prune the StripeSet of the disjunction vertex as $disj.\mathcal{SS} = \mathsf{prune\_stripes}(disj.\mathcal{SS}, c.\mathcal{SS}, child) \cup \mathsf{prune\_stripes}(disj.\mathcal{SS}, d.\mathcal{SS}, child)$. Now, that we have pruned Stripeset for vertex $disj$, we prune StripeSet of vertex a with respect to its adjacent vertices $disj$ and b by applying pruning rule 2:

$a.\mathcal{SS} = \mathsf{prune\_stripes}(a.\mathcal{SS}, disj.\mathcal{SS}, \perp) \cap \mathsf{prune\_stripes}(a.\mathcal{SS}, b.\mathcal{SS}, child)$. □

Another important issue is that a successful Stripe pruning operation at StripeSet $u.\mathcal{SS}$ of vertex u, introduces further pruning opportunities to all of its adjacent vertices (regardless edge direction). If v is an adjacent vertex of u, and StripeSet of v is also successfully pruned (*w.r.t.* some other adjacent vertex, for instance z) then we also need to check back at vertex u whether a new pruning opportunity has risen. For instance, consider the first example query q_1 and its Stripe Graph in Figure 2.2. Consider now that (the StripeSet of) vertex a_4 is successfully pruned *w.r.t.* (the StripeSet of) vertex a_2 through edge $(a_2 \rightarrow a_4)$. Possible pruning opportunities arise for pruning vertices

Algorithm 2.2: prune($Edge : e$)

```
Data: e : u -a-> v
Result: Stripe pruning of v.SS
1 begin
2     if (v.type ≠ disj) then
3         v.SS ←
          prune_stripes(v.SS, u.SS, a);
4     visited(v, u);
5     if (exhausted(v)) then
6         prune_stripes_disj(v);
7         forall e' ∈ out_edges(v) do
8             prune(e');
9 end
```

Figure 2.6: Stripe Pruning algorithm

Algorithm 2.3: validate($Edge : e$)

```
Data: e : u -a-> v
Result: validation of v.SS
1 begin
2     v.SS ←
      prune_stripes(v.SS, u.SS, a);
3     visited(v, u);
4     if (exhausted(v)) then
5         forall e' ∈ out_edges(v) do
6             validate(e')
7 end
```

Figure 2.7: Stripe Validation algorithm

a_3, a_5 and a_7 *w.r.t.* a_4, in addition to other opportunities that will further arise if any of those attempts actually prunes any Stripes. Later, if we try pruning the same vertex a_4 *w.r.t.* vertex a_3, new pruning opportunities arise for vertices a_2, a_5 and a_7. In essence, when a StripeSet pruning occurs at any graph vertex, then it can affect all other vertices in graph G. This effectively introduces graph cycles.

In order to avoid redundant or repeating pruning operations we employ the following pruning strategy: We divide Stripe pruning process in two distinct phases: During the first phase, called the *Pruning Phase*, we prune Stripes visiting vertices *only* in a bottom-up fashion without triggering further pruning to any adjacent vertices in the opposite direction. At the second phase, the *Validation Phase*, we repeat the process only visiting vertices as in a top-down traversal. This strategy effectively prunes all vertices with two complete graph traversals, eliminating any cycle traversals.

2.3.3.1 Pruning Phase

The pruning phase visits vertices of the Stripe Graph in a bottom up fashion. This can be considered as a top-down traversal of the inverse graph of the Stripe Graph *i.e.,* a graph $G' = (V, E')$ where $E' = \{ e' \,|\, e \in E \wedge e' = \mathsf{inverse}(e) \}$ and the inverse edge of $(u \xrightarrow{a} v)$ is $(v \xrightarrow{a} u)$. The recursive algorithm for the pruning phase is shown in

Figure 2.6 and it shares the same traversal method as Stripe projection in the sense that traversal will proceed only after a vertex is exhausted *i.e.,* has been visited from all incoming edges (line 5). The difference, of course, lies in that the direction of edges is now inverted, simulating a bottom-up traversal of the original Stripe Graph G. As in the case of the projection algorithm, this ensures that each vertex is fully processed (in this case, prune StripeSet *w.r.t.* all incoming edges) before being the new point of reference for any of its outgoing edges. However, this algorithm's main characteristic is the *lazy* pruning approach for disjunction vertices: When vertex v is reached from vertex u, we prune $v.\mathcal{SS}$ *w.r.t.* $u.\mathcal{SS}$ only if vertex v is **not** a disjunction vertex. (lines 2-3). In the case of a disjunction vertex, we perform no pruning but delay the pruning process until vertex v is exhausted. Only then do we prune StripeSet with respect to all adjacent vertices as defined in pruning rule 3 (function prune_stripes_disj, line 6).

2.3.3.2 Validation Phase

The validation phase is the pruning phase's dual operation. All pruning operations in bottom-up order have already been processed and if any successful Stripe pruning has occurred, we need to check (validate) whether further pruning can occur in the opposite direction. For that reason, the validation phase operates on the original Stripe Graph G. The validation algorithm executes exactly as the projection algorithm only that the statement in line 2 of Figure 2.3 (project_stripe, is replaced by a (prune_stripes) function call in order to process the remaining StripeSets that were ignored during the pruning phase. Again, the traversal method used in the Stripe projection algorithm is preserved so that the process continues only after a certain vertex is exhausted from its incoming vertices.

Example 2.5:. We now describe Stripe pruning for query q_2 which is shown in Figure 2.8. Steps (a) to (d) illustrate the pruning phase operating on the inverse Stripe Graph, while steps (e) - (h) illustrate the validation phase (on the original Stripe Graph). The pruning phase begins from vertex c and we first visit disjunction vertex, say *disj* (Figure 2.8(a)). As vertex *disj* is a disjunction vertex, no pruning is performed. In addition, since it is not yet exhausted (not yet visited from all incoming edges), the process backtracks, starting again from vertex d. Again we visit disjunction vertex *disj* and no pruning is performed. However, the visiting vertex is now exhausted and therefore we prune its Stripeset *w.r.t.* to the union of the StripeSets of its adjacent vertices c and d, using function prune_stripes_disj (Figure 2.8(b)). The process continues

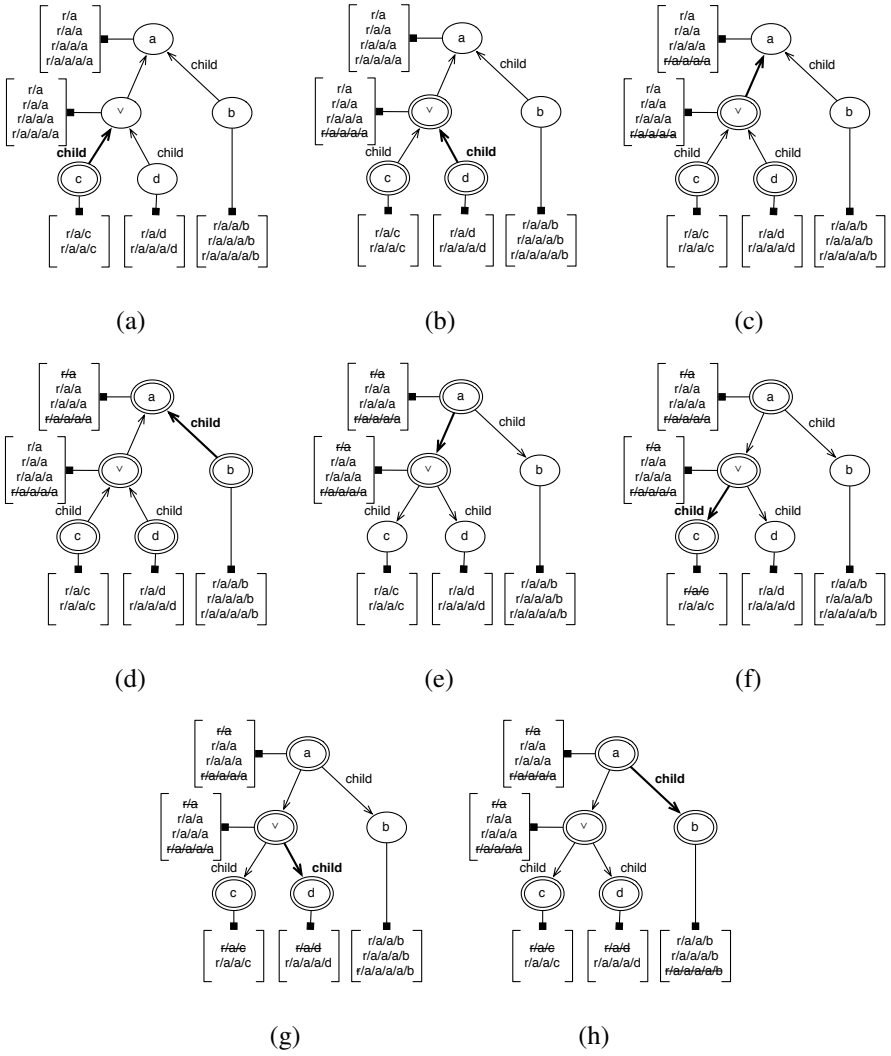


Figure 2.8: Pruning example for query q_2 : *descendant*::$a[c$ **or** $d]/b$. (a) - (d) Bottom-up Pruning on inverse Graph G' (e) - (h) Top-down Validation on original Graph G.

by traversing edge $(disj \rightarrow a)$ and pruning vertex a accordingly (Figure 2.8(c)). Vertex a is in turn not exhausted and thus the process backtracks again, starting over for vertex b and edge $(b \xrightarrow{child} a)$, pruning vertex a *w.r.t.* b (Figure 2.8(d)). The pruning phase is now completed and the validation phase begins from vertex a, pruning the StripeSet of *disj w.r.t.* a (Figure 2.8(e)). Likewise, the StripeSets of vertices c and d are pruned *w.r.t. disj* (Figure 2.8(f) , Figure 2.8(g)) while finally, vertex b is pruned *w.r.t.* a (Figure 2.8(h)). □

As in Stripe projection, each of the pruning and validation phases of the pruning process performs a full traversal of the input Stripe Graph. Thus, each of them takes $O(|V| + |E|)$ time in order to prepare (graph initialisation) and visit all vertices. Let

t_{prn} be the running time for function prune_stripes. We then have that the running time of Stripe pruning is $O(2(|V|+|E|t_{prn}))$.

The complete *Input Minimisation* process is presented in Figures 2.9 and 2.10, while the auxiliary functions being used are summarised in Table 2.6. First, the Stripe Dependency Graph is constructed for a given query and the Stripe projection process is invoked. As soon as it completes, all initial StripeSets are identified for each vertex of the Stripe Graph, and the pruning phase of the pruning process is invoked that initially prunes StripeSets in a bottom-up manner. Finally, the validation phase is called on the original Stripe Graph for any remaining StripeSet pruning. The overall running time of Algorithm 2.4: minimise_input is the time for the projection and pruning processes, thus: $O(3|V|+(t_{prj}+2t_{prn})|E|)$. Note that the projection process can check whether Stripe pruning can take place at any of the reference vertices while projects StripeSets for its visiting vertices. If no pruning is necessary for any of its visiting vertices then Stripe pruning process is skipped, limiting the overall Input Minimisation running time to $O(|V|+|E|t_{prj})$.

2.4 Path Minimisation

We have introduced a striped data model as a means to logically group XML nodes that share the same label-path. In this section, we define query equivalences that apply over the striped model and minimise the number of location step expressions of a query $q \in \mathcal{XP}$. The proposed equivalences hold between path expressions of the form: $Q : path/step_1/step_2$ and $Q' : path/step_2$. Location step expressions $step_1$, $step_2$ are defined as:

$$step_1 ::= axis :: nodetest$$
$$step_2 ::= step_1 \mid step_1[\,qualif\,]$$

while expressions *path*, *axis*, *nodetest* and *qualif* are as defined in Section 2.1.2. These equivalences have the following properties. Let q, q' be expressions of Q and Q' respectively. We have that:

1. Expression q' contains a location step expression less in comparison to expression q

Algorithm 2.4: minimise_input($\mathcal{XP} : q$)

Result: minimal I_q

```
1 begin
2     G ← dep_graph(q);
3     traverse(G, PRJ);
4     G ← inv_graph(G);
5     traverse(G, PRN);
6     G ← inv_graph(G);
7     traverse(G, VALD);
8 end
```

Algorithm 2.5: traverse(DAG : $G, Mode : m$)

```
1 begin
2     forall u ∈ V ∧ in_degree(u) = 0 do
3         forall e ∈ out_edges(u) do
4             traverse(e, m);
5 end
```

Algorithm 2.6: traverse($Edge : e, Mode : m$)

Data: $e = (u \xrightarrow{a} v)$

```
1 begin
2     process(e, m);
3     visited(e);
4     if (exhausted(v)) then
5         post_exhausted_process(v, m);
6         forall e' ∈ out_edges(v) do
7             traverse(e', m);
8 end
```

Figure 2.9: Input Minimisation algorithm (a)

Algorithm 2.7: process($Edge : e, Mode : m$)

Data: $e = (u \xrightarrow{a} v)$

```
1 begin
2     if (m = PRJ) then
3         v.SS ← v.SS ∪ project_stripes(u.SS, v.n, a);
4     else if (m = PRN) then
5         if (v.type ≠ disj) then
6             v.SS ← prune_stripes(v.SS, u.SS, a);
7     else if (m = VALD) then
8         v.SS ← prune_stripes(v.SS, u.SS, a);
9 end
```

Algorithm 2.8: post_exhausted_process($Vertex : v, Mode : m$)

```
1 begin
2     if (m = PRN ∧ v.type = disj) then
3         SS ← {};
4         forall e ∈ in_edges(v) do
5             SS ← SS ∪ prune_stripes(v.SS, u.SS, a);
6         v.SS ← v.SS ∩ SS;
7 end
```

Figure 2.10: Input Minimisation algorithm (b)

dep_graph	: $\mathcal{XP} \rightarrow$ DAG
inv_graph	: DAG $\rightarrow$ DAG
project_stripes	: $Set(Stripe) \times NodeTest \times Axis \rightarrow Set(Stripe)$
prune_stripes	: $Set(Stripe) \times Set(Stripe) \times Axis \rightarrow Set(Stripe)$
visited	: $Edge$
exhausted	: $Vertex \rightarrow Boolean$
in_edges	: $Vertex \rightarrow Set(Edge)$
out_edges	: $Vertex \rightarrow Set(Edge)$
in_degree	: $Vertex \rightarrow Integer$

Table 2.6: Input Minimisation Auxiliary Functions

2. Expression q is contained in expression q' ($q \subset q'$,). This means that the evaluation of expression q over a document T results in a NodeSet that is a proper subset of the NodeSet resulting from the evaluation of expression q' over the same document T. This is written as $\mathcal{S}[\![q]\!] \subset \mathcal{S}[\![q']\!]$.

Since $q \subset q'$ holds, expressions q and q' are *not* equivalent. And while this is generally true, expressions q and q' become equivalent when these are evaluated over the projected StripeSets, produced (as described in Section 2.3.2) for expression q. By recursively applying the proposed path equivalences in q, it is possible (subject to query q) to shrink it up to a single step expression. We term this process *Path Expression Minimisation*, or simply *Path Minimisation* and it depends on path equivalences over our striped data model. We now provide a simple example, demonstrating the path minimisation process:

Example 2.6: Consider path expressions $q = /child::a/child::d$ and $q' = /descendant::d$ and the document tree T shown in Figure 2.5(a). For queries q, q', we have that $\mathcal{S}[\![q]\!] = \{d_1\}$ and $\mathcal{S}[\![q']\!] = \{d_1, d_2\}$. As a result, expressions q and q' are not equivalent and in particular, $q \subset q'$ when evaluated over document T. Now, let us consider the projection process for expression q. The Stripe Graph for q is: $r \xrightarrow{child} a \xrightarrow{child} d$ while the *Result StripeSet* of q is: $\mathcal{RS}_q = \{S_{r/a/d}\}$, which defines the domain of all nodes that can be selected by expression q. Now consider evaluating expression q' over document T but restricting the result to the domain of all nodes that can be selected by expression q, *i.e.,* $\mathcal{RS}_q$. The evaluation of q' would then result in the same NodeSet as with the evaluation of q. □

We now formalise this observation as follows:

Definition 4.1: *We define a function* $\mathcal{F} : Pattern \rightarrow Node \times Set(Stripe) \rightarrow Set(Node)$ *as the composition of two functions* $\mathcal{S} : Pattern \rightarrow Node \rightarrow Set(Node)$ *and* $\mathcal{V} : Set(Node) \times Set(Stripe) \rightarrow Set(Node)$. *Function* $\mathcal{S}$ *is defined as in the formal semantics for XPath expressions while function* $\mathcal{V}$ *validates that the result of* $\mathcal{S}$ *is contained in the set of nodes produced by a given StripeSet. Two path expressions q,* q' *with* $q \subset q'$ *are equivalent over the striped model, written as* $q \overset{\text{SRX}}{\equiv} q'$, *iff* $\mathcal{F}[\![q]\!](x, \mathcal{RS}_q) = \mathcal{F}[\![q']\!](x, \mathcal{RS}_q)$ *for any context node x and with* $\mathcal{RS}_q$ *being the Resulting StripeSet of q.* □

From the definition of $\mathcal{RS}$ (Equation (2.2)), we have that for any expression q, the following holds:

$$\mathcal{F}[\![q]\!](x, \mathcal{RS}_q) = \mathcal{S}[\![q]\!](x)$$

since the Result StripeSet $\mathcal{RS}_q$ will always contain all Stripes that in turn contain all possible result nodes for expression q. Thus, the above condition for path expression equivalence can be reduced to:

$$q \overset{srx}{\equiv} q' \text{ iff } \mathcal{S}[\![q]\!](x) = \mathcal{F}[\![q']\!](x, \mathcal{RS}_q)$$

Lemma 4.1: *If there exist two path expressions q and q' with $q \overset{srx}{\equiv} q'$, then expression q can be substituted by expression q'.* □

We now provide a listing of equivalences that hold over the striped model. These equivalences are of the general form $path/step_1/step_2 \overset{srx}{\equiv} path/step_2$. We merely list equivalences where expression $step_2$ is of the form of $step_1 ::= axis :: nodetest$ but the same equivalences apply when $step_2$ expressions are of the form: $axis :: nodetest[qualif]$. Expressions n and m are of type *NodeTest*. According to the axis of the removed location step of type $step_1$, we define the following equivalences over the striped model:

***Self* axis Equivalences**:

$$path/self::n/self::m \overset{srx}{\equiv} path/self::m \tag{2.3}$$
$$path/self::n/child::m \overset{srx}{\equiv} path/child::m \tag{2.4}$$
$$path/self::n/descendant::m \overset{srx}{\equiv} path/descendant::m \tag{2.5}$$
$$path/self::n/descendant\text{-}or\text{-}self::m \overset{srx}{\equiv} path/descendant\text{-}or\text{-}self::m \tag{2.6}$$
$$path/self::n/following::m \overset{srx}{\equiv} path/following::m \tag{2.7}$$
$$path/self::n/following\text{-}sibling::m \overset{srx}{\equiv} path/following\text{-}sibling::m \tag{2.8}$$

***Child* axis Equivalences**:

$$path/child::n/self::m \overset{srx}{\equiv} path/child::m \tag{2.9}$$
$$path/child::n/child::m \overset{srx}{\equiv} path/descendant::m \tag{2.10}$$
$$path/child::n/descendant::m \overset{srx}{\equiv} path/descendant::m \tag{2.11}$$
$$path/child::n/descendant\text{-}or\text{-}self::m \overset{srx}{\equiv} path/descendant::m \tag{2.12}$$

Descendant **axis Equivalences**:

$$path/descendant::n/self::m \overset{srx}{\equiv} path/descendant::m \quad (2.13)$$

$$path/descendant::n/child::m \overset{srx}{\equiv} path/descendant::m \quad (2.14)$$

$$path/descendant::n/descendant::m \overset{srx}{\equiv} path/descendant::m \quad (2.15)$$

$$path/descendant::n/descendant\text{-}or\text{-}self::m \overset{srx}{\equiv} path/descendant::m \quad (2.16)$$

Descendant-or-self **axis Equivalences**:

$$path/descendant\text{-}or\text{-}self::n/self::m \overset{srx}{\equiv} path/descendant\text{-}or\text{-}self::m \quad (2.17)$$

$$path/descendant\text{-}or\text{-}self::n/child::m \overset{srx}{\equiv} path/descendant::m \quad (2.18)$$

$$path/descendant\text{-}or\text{-}self::n/descendant::m \overset{srx}{\equiv} path/descendant::m \quad (2.19)$$

$$path/descendant\text{-}or\text{-}self::n/descendant\text{-}or\text{-}self::m \overset{srx}{\equiv} path/descendant\text{-}or\text{-}self::m \quad (2.20)$$

Parent **axis Equivalences**:

$$path/parent::n/self::m \overset{srx}{\equiv} path/parent::m \quad (2.21)$$

$$path/parent::n/parent::m \overset{srx}{\equiv} path/ancestor::m \quad (2.22)$$

$$path/parent::n/ancestor::m \overset{srx}{\equiv} path/ancestor::m \quad (2.23)$$

$$path/parent::n/ancestor\text{-}or\text{-}\mathrm{self}::m \overset{srx}{\equiv} path/ancestor::m \quad (2.24)$$

Ancestor **axis Equivalences**:

$$path/ancestor::n/self::m \overset{srx}{\equiv} path/ancestor::m \quad (2.25)$$

$$path/ancestor::n/parent::m \overset{srx}{\equiv} path/ancestor::m \quad (2.26)$$

$$path/ancestor::n/ancestor::m \overset{srx}{\equiv} path/ancestor::m \quad (2.27)$$

$$path/ancestor::n/ancestor\text{-}or\text{-}\mathrm{self}::m \overset{srx}{\equiv} path/ancestor::m \quad (2.28)$$

***Ancestor-or-self* axis Equivalences**:

$$path/ancestor\text{-}or\text{-self}::n/self::m \overset{\text{srx}}{\equiv} path/ancestor\text{-}or\text{-self}::m \quad (2.29)$$

$$path/ancestor\text{-}or\text{-self}::n/parent::m \overset{\text{srx}}{\equiv} path/ancestor::m \quad (2.30)$$

$$path/ancestor\text{-}or\text{-self}::n/ancestor::m \overset{\text{srx}}{\equiv} path/ancestor::m \quad (2.31)$$

$$path/ancestor\text{-}or\text{-self}::n/ancestor\text{-}or\text{-self}::m \overset{\text{srx}}{\equiv} path/ancestor\text{-}or\text{-self}::m \quad (2.32)$$

We now present the proof for Equivalence 2.10. All proofs can be found at Chapter A.

Equivalence 2.10: For any path expression *path*, the following holds:

$$path/child::n/child::m \overset{\text{srx}}{\equiv} path/descendant::m$$

Proof: Let $q = path/child::n/child::m$, $q' = path/descendant::m$ while $\mathcal{SS}_{path}$, $\mathcal{SS}_n$ and $\mathcal{SS}_m$ be the StripeSets projected for path expression *path* and for each of the location steps that follow in q. For path expression q, we have that: $\mathcal{RS}_q = \mathcal{SS}_m$. From Equation (2.1) we have that for any context node x:

$$\mathcal{S}[\![path]\!]x \subseteq \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \quad (2.33)$$

and thus:

$$y \in \mathcal{S}[\![path]\!]x \Rightarrow y \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \quad (2.34)$$

According to the Stripe projection process for *child* axis, we have that for StripeSets $\mathcal{SS}_n$, $\mathcal{SS}_m$ the following hold:

$$\begin{aligned}\mathsf{ss\_nodes}(\mathcal{SS}_n) &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge \\ &\qquad \mathsf{path}(x) = \mathsf{maximal\_prefix}(\mathsf{path}(y)) \wedge \tau(y,n) \,\} \\ &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge y \in \mathsf{children}(x) \wedge \tau(y,n) \,\} \quad (2.35) \\ \mathsf{ss\_nodes}(\mathcal{SS}_m) &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge \\ &\qquad \mathsf{path}(x) = \mathsf{maximal\_prefix}(\mathsf{path}(y)) \wedge \tau(y,m) \,\} \\ &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge y \in \mathsf{children}(x) \wedge \tau(y,m) \,\} \quad (2.36)\end{aligned}$$

From Equations (2.35) , (2.36) we have that:

$$\begin{aligned}\forall z \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \exists x,y : x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge \\ y \in \mathsf{children}(x) \wedge \tau(y,n) \wedge z \in \mathsf{children}(y) \wedge \tau(z,m) \quad (2.37)\end{aligned}$$

Now, for any context node x, we have:

$$\begin{aligned}
\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/descendant{::}m]\!](x,\mathcal{RS}_q)\\
&= \mathcal{V}(\mathcal{S}[\![path/descendant{::}m]\!]x,\mathcal{RS}_q)\\
&= \{\,x_3 \,|\, x_3 \in \mathcal{S}[\![path/descendant{::}m]\!]x \wedge x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q)\,\}\\
&= \{\,x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![descendant{::}m]\!]x_1 \wedge\\
&\qquad x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q)\,\}\\
&= \{\,x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{children}^+(x_1) \wedge \tau(x_3,m) \wedge\\
&\qquad x_3 \in \mathsf{ss_nodes}(\mathcal{SS}_m)\,\} \qquad (2.38)\\
&= \{\,x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{children}^+(x_1) \wedge \tau(x_3,m) \wedge\\
&\qquad x_1 \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}(x_1) \wedge \tau(x_2,n) \wedge\\
&\qquad x_3 \in \mathsf{children}(x_2) \wedge \tau(x_3,m)\,\} \qquad (2.39)\\
&= \{\,x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{children}(x_1) \wedge \tau(x_2,n) \wedge\\
&\qquad x_3 \in \mathsf{children}(x_2) \wedge \tau(x_3,m)\,\} \qquad (2.40)\\
&= \{\,x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![child{::}n]\!]x_1 \wedge x_3 \in \mathcal{S}[\![child{::}m]\!]x_2\,\}\\
&= \mathcal{S}[\![path/child{::}n/child{::}m]\!]x
\end{aligned}$$

We use Equation (2.37) to get from Equation (2.38) to (2.39). We then consider Equation (2.34) as well as that

$$x_2 \in \mathsf{children}(x_1) \wedge x_3 \in \mathsf{children}(x_2) \implies x_3 \in \mathsf{children}^+(x_1)$$

to discard all redundant conjuncts and get to Equation (2.40). □

We also provide a set of *conditional* equivalences, *i.e.,* path equivalences that hold for two path expressions q and q' over the striped model but only if a specified condition c also holds. We denote conditional equivalences as: $q \underset{c}{\overset{\mathsf{srx}}{\equiv}} q'$.

Lemma 4.2: *If there exist two path expressions q and q' with $q \underset{c}{\overset{\mathsf{srx}}{\equiv}} q'$, then expression q can be substituted by expression q', as long as condition c holds.* □

We now define the following set of conditional equivalences over the striped model:

Following **axis Equivalences**:

$$path/following::n/self::m \overset{\underset{c}{\text{srx}}}{\equiv} path/following::m \tag{2.41}$$

$$path/following::n/child::m \overset{\underset{c}{\text{srx}}}{\equiv} path/following::m \tag{2.42}$$

$$path/following::n/descendant::m \overset{\underset{c}{\text{srx}}}{\equiv} path/following::m \tag{2.43}$$

$$path/following::n/descendant\text{-}or\text{-}self::m \overset{\underset{c}{\text{srx}}}{\equiv} path/following::m \tag{2.44}$$

Preceding **axis Equivalences**:

$$path/preceding::n/self::m \overset{\underset{c}{\text{srx}}}{\equiv} path/preceding::m \tag{2.45}$$

$$path/preceding::n/child::m \overset{\underset{c}{\text{srx}}}{\equiv} path/preceding::m \tag{2.46}$$

$$path/preceding::n/descendant::m \overset{\underset{c}{\text{srx}}}{\equiv} path/preceding::m \tag{2.47}$$

$$path/preceding::n/descendant\text{-}or\text{-}self::m \overset{\underset{c}{\text{srx}}}{\equiv} path/preceding::m \tag{2.48}$$

The condition c for which the above equivalences hold, is that for expression q : $path/a_1::n/a_2::m$, the candidate nodes for expression $a_1::n$ must not be related to any of the candidate nodes for expression $path$, with the ancestor relationship. Thus, the following holds:

$$\forall x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \nexists y : y \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge y \in \mathsf{parent}^+(x) \tag{2.49}$$

We now present the proof for Equivalence 2.43. All proofs can be found at Chapter A.

Equivalence 2.43: For any path expression $path$, the following holds:

$$path/following::n/descendant::m \overset{\underset{c}{\text{srx}}}{\equiv} path/following::m$$

Proof: Let $q = path/following::n/descendant::m$, $q' = path/following::m$ while $\mathcal{SS}_{path}$, $\mathcal{SS}_n$ and $\mathcal{SS}_m$ be the StripeSets projected for path expression $path$ and for each of the location steps that follow in q. According to the Stripe projection for *following* axis, we have that for StripeSet $\mathcal{SS}_n$ the following holds:

$$\mathsf{ss\_nodes}(\mathcal{SS}_n) = \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge y \in \mathsf{children}^+(\mathsf{root}(x)) \wedge \tau(y,n) \,\} \tag{2.50}$$

as the Stripe projection for *following* axis, results in selecting **all** Stripes of the XML document. However, since condition c enforces Equation (2.49), we have that:

$$\begin{aligned}\mathsf{ss\_nodes}(\mathcal{SS}_n) = \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge y \in \mathsf{children}^+(\mathsf{root}(x)) \\ \wedge\, \tau(y,n)\,\} \wedge y \notin \mathsf{parent}^+(x)\end{aligned} \quad (2.51)$$

According to the Stripe projection for *descendant* axes, we have that:

$$\begin{aligned}\mathsf{ss\_nodes}(\mathcal{SS}_m) &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge \mathsf{path}(x) \in \mathsf{proper\_prefix}(\mathsf{path}(y)) \wedge \tau(y,m)\,\} \\ &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge y \in \mathsf{children}^+(x) \wedge \tau(y,m)\,\}\end{aligned} \quad (2.52)$$

From Equations (2.51) , (2.52) we have that:

$$\begin{aligned}&\forall z \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \exists x, y : x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge \\ &y \in \mathsf{children}^+(\mathsf{root}(x)) \wedge \tau(y,n) \wedge y \notin \mathsf{parent}^+(x) \wedge z \in \mathsf{children}^+(y) \wedge \tau(z,m)\end{aligned} \quad (2.53)$$

$$\begin{aligned}\mathcal{F}[\![q']\!](x, \mathcal{RS}_q) &= \mathcal{F}[\![path/following{::}m]\!](x, \mathcal{RS}_q) \\ &= \mathcal{V}(\mathcal{S}[\![path/following{::}m]\!]x, \mathcal{RS}_q) \\ &= \{\, x_3 \,|\, x_3 \in \mathcal{S}[\![path/following{::}m]\!]x \wedge x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q)\,\} \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![following{::}m]\!]x_1 \wedge \\ &\qquad x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q)\,\} \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{following}(x_1) \wedge \tau(x_3,m) \wedge \\ &\qquad x_3 \in \mathsf{ss\_nodes}(\mathcal{SS}_m)\,\} \qquad (2.54) \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{following}(x_1) \wedge \tau(x_3,m) \wedge \\ &\qquad x_1 \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}^+(\mathsf{root}(x_1)) \wedge \tau(x_2,n) \wedge \\ &\qquad x_2 \notin \mathsf{parent}^+(x_1) \wedge x_3 \in \mathsf{children}^+(x_2) \wedge \tau(x_3,m)\,\} \qquad (2.55) \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge \\ &\qquad (((x_2 \in \mathsf{following}(x_1) \vee x_2 \in \mathsf{parent}^+(x_1)) \wedge \tau(x_2,*)) \wedge \\ &\qquad x_3 \in \mathsf{children}^+(x_2) \wedge \tau(x_3,m)) \wedge \\ &\qquad x_1 \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}^+(\mathsf{root}(x_1)) \wedge \tau(x_2,n) \wedge \\ &\qquad x_2 \notin \mathsf{parent}^+(x_1) \wedge x_3 \in \mathsf{children}^+(x_2) \wedge \tau(x_3,m)\,\} \qquad (2.56) \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{following}(x_1) \wedge \tau(x_2,n) \wedge \\ &\qquad x_3 \in \mathsf{children}^+(x_2) \wedge \tau(x_3,m)\,\} \qquad (2.57)\end{aligned}$$

$$
\begin{aligned}
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![following::n]\!]x_1 \wedge \\
&\qquad x_3 \in \mathcal{S}[\![descendant::m]\!]x_2 \,\} \\
&= \mathcal{S}[\![path/following::n/descendant::m]\!]x
\end{aligned}
$$

We use Equation (2.53) to get from Equation (2.54) to (2.55). In Equation (2.56), we rewrite $x_3 \in \mathsf{following}(x_1) \wedge \tau(x_3, m)$ as

$$(((x_2 \in \mathsf{following}(x_1) \vee x_2 \in \mathsf{parent}^+(x_1)) \wedge \tau(x_2, *)) \wedge x_3 \in \mathsf{children}^+(x_2) \wedge \tau(x_3, m)$$

to introduce the ancestor-descendant relationship between x_2 and x_3. Then, we discard all redundant terms to get to Equation (2.57). □

Note that we can infer whether condition c holds or not from the projected StripeSets $\mathcal{SS}_{path}$ and $\mathcal{SS}_n$. In particular, if there does not exist a Stripe in $\mathcal{SS}_n$ so that its path is a proper prefix of the path of any of the Stripes in $\mathcal{SS}_{path}$, then we know that there is no node $y \in \mathsf{ss\_nodes}(\mathcal{SS}_n)$ that is an ancestor node of any node $x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path})$.

2.5 Related Work

We described Striping, a decomposition method that stores XML nodes according to their label-path. Striping is reminiscent of *vertical partitioning*, a decomposition technique proposed at the 70s for the relational model (*e.g.,* [12]). In traditional relational databases, all attributes of a tuple are clustered together. This implies that to access a certain attribute, the whole tuple must be read from disk and placed into the buffer pool. Vertical partitioning employs a column-based approach (as opposed to the row-based approach of a traditional DBMS), where a relation is partitioned into a number of column relations, one for each relation attribute. In [31], a column relation is described as a binary relation containing a tuple surrogate and one attribute. This has formed the basis for modern column-based DBMSs such as Sybase IQ (*e.g.,* [68]), MonetDB [16] and C-Store [93]. A column-based decomposition usually provides better query performance compared to the traditional row-based storage as a relatively small number of attributes is usually involved in a query. In these cases, vertical partitioning provides access directly to the attributes needed for query evaluation and thus minimises query input. Striping can be considered as the application of vertical partitioning to the XML model. Since XML data is schema-less, we partition the XML tree structure according to label-paths. This way, XML data that are semantically related are clustered together in a single Stripe.

In the context of XML, it is shown that in most cases only a small percentage of the original XML document is needed to evaluate a given XML query [70]. With the striped XML model, we strive to select the parts of the document that are only relevant to a given query. Label-paths play a central role in XML node addressing, especially in the absence of an XML schema. The importance of selecting XML nodes based on their label-paths was evident even from the first approaches for storing and querying XML data using a relational database [104, 58]. In addition, a similar model was later adopted in [44] for efficient XPath evaluation. This approach is based on a fixed relational schema (mapping), where the label-paths of XML nodes were encoded and explicitly stored. The selection of XML nodes during query evaluation is handled by the evaluation of regular expressions and join operators on the label-path attributes of the fixed schemas.

An earlier approach to XML storing and querying is the Edge approach [40], where a fixed relational schema is also used, but, as its name suggests, merely stores parent-child node relationships (edges). In this case, the selection of XML nodes is handled by self-join operations on the Edge relation. Experimental results show that, compared to the former approach that utilises label-paths as a means to locate XML nodes, the Edge approach underperforms in most cases as it employs a large number of self-join operators on the Edge relation for selecting the appropriate XML nodes.

A refinement of the Edge approach is the Monet XML model [87]. Similar to our decomposition model, Striping, XML data is partitioned according to all possible label-paths. The Monet decomposition can be seen as the partition of the Edge relation to a number of binary relations, each of which stores node pairs (edges) for a certain label-path. The selection of XML nodes during query evaluation is now made explicitly by accessing the binary relations that contain data relevant to the query.

The idea of Striping has also been proposed for the holistic evaluation of twig pattern queries [24, 25]. In this work, a Streaming model is proposed, the Prefix Path Streaming (PPS), which divides XML nodes in Streams (Stripes) based on their prefix path (label-path). For the first time, the notion of selecting useful Streams for query evaluation is introduced and a recursive Stream pruning algorithm is proposed that, similar to the proposed Input Minimisation process, eliminates useless Streams that do not contain query results. Nevertheless, this is constrained to twig pattern queries that merely contain parent-child and ancestor-descendant node relationships and pattern branches having conjunct semantics.

As already described, the general idea of Striping for storing XML data has been

conceived and used since the early years that XML has emerged. However, it is only recently that it received attention for query optimisation. In [11], the authors study in depth the benefits of Striping (termed path-partitioning), and exploit them to optimise query processing. To that end, a query pattern minimisation process is proposed that eliminates query pattern nodes that are not important to the query output. This is accomplished by the use of a path summary, a structure that captures all unique root-to-leaf label-paths and enables the selection of useful (to a pattern query) paths, termed "relevant paths". The query pattern minimisation process has the same effect as the proposed Input and Path Minimisation processes. Nevertheless, as in the case of [24, 25], the path selection language which is considered merely considers containment relationships, *i.e.,* parent-child and ancestor-descendant node relationships.

Chapter 3

Query Evaluation over a Striped Model

XPath is an expression language defined by the World Wide Web Consortium [5], that is used primarily for selecting nodes of an XML tree. Location step expressions provide the ability to navigate the tree structure in any direction, selecting nodes based on the expression semantics.

The current version of the XPath language is XPath 2.0 [14], which became a recommendation in 2007. It has significantly evolved since its predecessor, XPath 1.0 [32], mainly due to the new adopted data model, XDM [101], in which every value is a sequence of items. To that end, XPath is a powerful sequence processing language, in addition to being an expression language for selecting XML tree nodes. The evaluation of XPath expressions efficiently is of paramount importance for the XML world as it is extensively used as a sub-language for processing XSL transformations and XQuery expressions:

XSLT The "eXtensible Stylesheet Language Transformations" is a language used for transforming XML documents into other XML documents. The original document remains unchanged, while a new document is created based on the content of the original one and a defined template, a stylesheet. In XSLT 2.0 [62], XPath 2.0 is used for identifying the part of the original document that is later processed to form the new document.

XQuery This is the procedural query language for processing collections of XML documents. It emerged as the means to provide flexible query facilities to data derived from diverse sources such as semi-structured documents, relational databases, object repositories that can be uniformly accessed using real and virtual XML documents. XPath 2.0 is actually a subset of XQuery 1.0 [92], sharing a common

data model, the XDM. Similarly to XSLT, XQuery uses XPath expressions to locate parts of input document(s) that are later processed according to the query expression.

The focus in this chapter is twofold: First, we provide the evaluation algorithms that our XML query engine considers for evaluating queries in $\mathcal{XP}$, a large fragment of XPath 2.0, described in Section 2.1.2. These algorithms are independent of the storage model and thus can be used by any query engine that requires access to lists of XML nodes. Later, we reason whether and under which circumstances, such evaluation algorithms can be enhanced for the striped storage model, presented in Section 2.2.

The rest of the chapter is organised as follows: In Section 3.1, we introduce preliminary information regarding XPath evaluation and the region encoding labelling scheme we regard from Stripe storage. In Section 3.2, we describe conceptual evaluation issues for evaluating an $\mathcal{XP}$ expression over the striped representation. In Section 3.3, we present our evaluation algorithms considered for evaluating $\mathcal{XP}$ expressions. We emphasise on algorithms that evaluate location step expressions and the way such are treated when they occur in predicate expressions. We then turn our attention on how and under what conditions such algorithms can be optimised when the striped representation is considered; we term this *Stripe-aware Optimisation*, described in Section 3.4. Finally, in Section 3.5, we discuss related work and in Section 3.6, we review our conclusions.

3.1 Preliminaries

In Section 2.1.2, we introduced some basic notions of XPath 2.0 and defined a fragment of it, $\mathcal{XP}$. We now provide more detail regarding evaluating XPath expressions. In addition, we remind the reader of the region encoding labelling scheme (Section 1.1.1.2), that has been extensively used for serialising XML documents, and how this relates to XPath expression evaluation.

3.1.1 XPath Evaluation

XPath 2.0 [14] is an expression language which aims to address the nodes of the tree representation of XML documents. XPath expressions mainly operate on XML trees but also on typed atomic values and sequences, *i.e.,* ordered collections of zero or more items, as defined in the XPath 2.0 Data Model (XDM) [101]. Sequences are

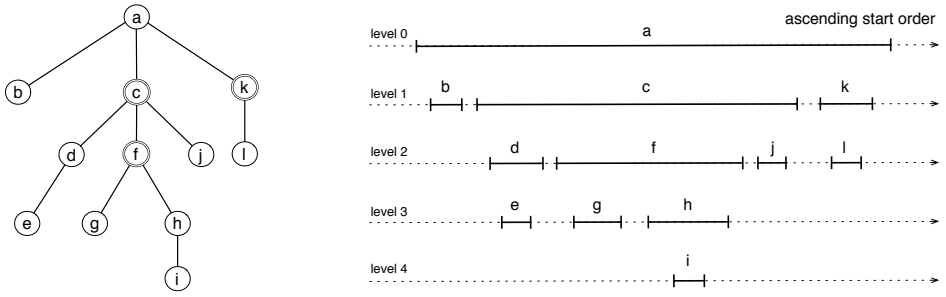
heterogeneous; a sequence item can either be an atomic value or a reference to an XML tree node. The result of an XPath expression is always a sequence.

As described in the XPath specification: "XPath gets its name from its use of a path notation for navigating through the hierarchical structure of an XML document". This introduces the core expression and most distinctive feature of XPath: the *path expression* $E_1/E_2/\ldots/E_n$ which is a sequence of expressions E_i where $i \in [1,n]$, used to locate XML nodes within trees. Such an expression is evaluated from left to right. Given the first node from a sequence S^{i-1}, the so called *context node*, the evaluation of sub-expression E_i yields a new node sequence S_1^i. After the evaluation of sub-expression E_i for all k nodes in sequence S^{i-1}, all the produced sequences $S_i^i, S_2^i, \ldots, S_k^i$ are then merged and duplicate nodes (based on node identity) are removed. The final sequence S^i is produced containing nodes in document order. Then, the produced sequence S^i for the i^{th} expression, provides the context node sequence for the evaluation of sub-expression E_{i+1}, *i.e.,* the evaluation of each sub-expression provides the context node sequence for the evaluation of the sub-expression that follows. The final sequence, produced for step n (which may contain atomic values) is the result for the path expression.

The dominant expression for tree node navigation is the *axis step* expression and in particular the *location step* $a::n$. A location step produces a sequence of nodes reachable from a context node and it is made of two parts: The *axis* part, a, which specifies the direction of the location step and the nodetest part, n, which is used for selecting nodes according to their type or name. An optional part of an axis step expression is a predicate list that follows a location step and filters its produced node sequence. The final sequence produced for the axis step, contains nodes produced by the location step that also satisfy all predicates of the predicate list, working from left to right.

3.1.2 Region Encoding

As already described in Section 1.1.1.2, labelling schemes address the problem of efficiently evaluating XML node structural relationships by maintaining the structural information of the tree nodes. Such relationships can then be tested by comparing the structural information of the tree nodes. For serialising XML nodes at Stripes, we use the region encoding scheme, augmented with parent node information. To that end, each node u of an XML tree is assigned a triplet: $\langle\, start, end, par \,\rangle$. The $start$ value is



(a) A small XML tree (b) The interval representation

Figure 3.1: Region-based interval representation of an XML tree

the value assigned to each node when encountered in a depth-first traversal of the XML tree, while the *end* is the value assigned when leaving a node during the same traversal. The *par* value stands for the *start* value of a node's parent. Since *start* is the pre-order rank of a node, it can also serve as a unique id for nodes. In addition, we have that for any two nodes u, v of the XML tree, u occurs *before* v iff $u.start < u.start$. Furthermore, for the forward step axes, namely *self*, *child*, *descendant*, *descendant-or-self*, *following* and *following-sibling* axes, the following hold:

- $v = u$ (implying the self axis), iff $v.start = u.start$
- v is a *descendant* of u, denoted $v \in \mathsf{children}^+(u)$, iff $u.start < v.start \wedge u.end \leq v.end$.
- v is a *descendant-or-self* of u, denoted $v \in \mathsf{children}^*(u)$, iff $v \in \mathsf{children}^+(u) \vee v = u \Leftrightarrow u.start \leq v.start \wedge u.end \leq v.end$
- v is a *child* of u, denoted $v \in \mathsf{children}(u)$, iff $v.par = u.start$
- v is a *following* of u, denoted $v \in \mathsf{following}(u)$, iff $v.start > u.end$
- v is a *following-sibling* of u, denoted $v \in \mathsf{following\text{-}sibling}(u)$, iff $v \in \mathsf{following}(u) \wedge v.par = u.par$

An example XML tree, along with its interval representation, is depicted in Figure 3.1. Each node's interval is based on its $(start, end)$ values as these are defined by the region encoding scheme. Note that containment node relationships (as defined in [105]) are faithfully maintained by node interval containment.

3.2 Evaluation Model

When XPath 2.0 [14] emerged as the successor of XPath 1.0 [32], the first thing someone would notice was the fundamental difference between their underlying data models. Besides the additional data types and build-in functions, the most important change was the move from the unordered node-sets supported in XPath 1.0 to the ordered item sequences[1] of XPath 2.0.

In XPath 1.0, in order to process a node collection, a node-set was used. Node-sets, being sets of nodes as their name suggests, had two main properties: (a) they were unordered and (b) they did not support duplicate nodes. Despite the fact, though, that node-sets were unordered, all nodes contained in a node-set were processed in a default order; the document order.

In XPath 2.0, the concept of the node-set has been extended to item sequences. First of all, sequences are heterogeneous collections, which means that they may contain atomic values as well as XML node references. In addition, sequences are ordered and may contain duplicates. In XPath 2.0, the default order used to process a node collection (i.e. a node sequence) is not necessarily the document order but, rather, the sequence order. However, XPath 2.0 expressions have to maintain backward compatibility with XPath 1.0. To that end, path expressions are defined so that they always return duplicate-free node sequences in document order, emulating XPath 1.0 node-sets.

3.2.1 Stripe Abstraction

For emulating node sequences, we use the Stripe abstraction. A *Stripe* is an ordered collection of XML nodes that allows duplicates. The Stripe order is implicit. To capture the structural information of XML nodes in addition to their type and content, we encode them as tuples of the form: $\langle start, end, par, level, kind, label, text \rangle$. The *start*, *end* and *par* values make up their region encoding, while the *level* value is the node's level in the tree. The *kind* value specifies the node kind and the *label* value corresponds to the the element label or attribute name. In the case of leaf XML nodes, the actual text (PCDATA or attribute value) that the node refers to is stored as the *text* value.

We also define three properties that a Stripe may or may not satisfy: The doc_order, the fwd and the single_level properties.

[1] We mostly regard node sequences in this work, although an item sequence is heterogeneous, *i.e.,* it may contain items of different types.

The doc_order **property** The *start* value of an XML node signifies its pre-order visit in the XML tree. When the Stripe order coincides with the ascending order of the nodes of the Stripe based on their *start* value, we say that the Stripe is in document order or that it satisfies the doc_order property. Formally, when the doc_order property holds for Stripe S, then for any pair of successive nodes (n_i, n_{i+i}) in S, we have that $n_i \ll n_{i+1} \Leftrightarrow n_i.start < n_{i+1}.start$.

The fwd **property** When each node of a Stripe *follows* all nodes that exist before that in the Stripe, we say that Stripe satisfies the fwd property. Formally, when the fwd property holds for Stripe S, then for any pair of successive nodes (n_i, n_{i+i}) in S, we have that $n_{i+1} \in \text{following}(n_i) \Leftrightarrow n_i.end < n_{i+1}.start$. This shows that the fwd property also implies the doc_order property since for each node n_i, $n_i.start \leq n_i.end$ always holds by definition.

The single_level **property** As the name suggests, when a Stripe satisfies the single_level property, it contains a collection of nodes that occur at the same tree level. Formally, when the single_level property holds for Stripe S, then for any pair of nodes (n_i, n_k) in S, we have that $n_i.level = n_k.level$. When a Stripe satisfies both the doc_order and single_level properties then it also satisfies the fwd property.

3.2.2 Operators

We now define some basic operations over Stripes for handling XPath expressions. All operators accept one or more Stripes as input and produce a single Stripe as output.

Step A location step or simply *step* operator $step(L,R)$ is the most basic operation for node navigation. It captures the semantics of a location step expression $a::n$ by operating on two input Stripes L and R and according to axis a of the location step. Recall that the nodetest n semantics are captured by the Stripe Projection process, described in Section 2.3. Stripe L provides the context node sequence for the evaluation of the location step, while the Stripe R provides a candidate node sequence from which the final result sequence will be produced, according to axis a.

Filter A *filter* operator $flt(L,R)$, as its name suggests, is a filtering operation capturing the semantics of a predicate expression. Similarly to the step operator, it operates on two input Stripes L and R. Stripe L provides the input sequence which is

filtered based on the effective boolean value (as defined in [14] for sequences) of the (partial) node sequence produced from the evaluation of the predicate expression (Stripe R).

Selection A *selection* operator $select(S)$ operates on the $text$ value of the nodes of its input Stripe. It accepts a value-based predicate and outputs a Stripe with those nodes whose $text$ values satisfy the given predicate.

Set Operators There are three self-described set operators: The set *union* operator $union(L,R)$, the set *intersection* operator $intersect(L,R)$ and the set *difference* operator $except(L,R)$ that produce the union, intersection and set difference of the two input Stripes L and R respectively.

Boolean Operators Conjunction, disjunction and negation boolean operators are defined that operate on the effective boolean value of input Stripes. The result is either an empty Stripe, casting to the boolean value $false$, or a non-empty Stripe which casts to the boolean value $true$.

Scan A *scan* operator is a lower-level operator for accessing raw data. It may have a single Stripe or multiple Stripes as input. In either case, it produces a Stripe containing all nodes from its input Stripe(s) in document order, *i.e.,* the resulting Stripe always satisfies the doc_order property.

These operators provide the building blocks for evaluating XPath expressions. As already described, each operator accepts a number of input Stripes and produces an output Stripe. Input Stripes may satisfy a set of properties. It is beneficial to know whether these properties are also retained by the output Stripes of the operators defined above.

We have already described that the doc_order property holds for the output of scan operators. We extensively discuss this property later, when we describe the evaluation algorithms in Section 3.3. Regarding the other two Stripe properties, fwd and single_level, we provide a set of rules identifying whether such properties hold for the output Stripe of an operator. These rules delegate the process to a combination of their input Stripes and according to the operator semantics. To that end, we define a general function $\mathsf{f} : Stripe \rightarrow boolean$ that accepts a Stripe argument and returns whether f property satisfies the argument Stripe or not. Function f() is overloaded to also accept operator arguments, since the result of an operator is always a Stripe. The rule set for both the fwd and single_level properties is summarised in Table 3.1.

$\mathsf{f}(step(L,R))$	$=$	$\mathsf{f}(R)$
$\mathsf{f}(flt(L,R))$	$=$	$\mathsf{f}(L)$
$\mathsf{f}(union(L,R)$	$=$	$\mathsf{f}(scan(L,R))$
$\mathsf{f}(intersection(L,R))$	$=$	$\mathsf{f}(L) \vee \mathsf{f}(R)$
$\mathsf{f}(except(L,R))$	$=$	$\mathsf{f}(L)$
$\mathsf{f}(select(S))$	$=$	$\mathsf{f}(S)$
$\mathsf{f}(scan(S))$	$=$	$true$
$\mathsf{f}(scan(S_1,\ldots,S_n))$	$=$	$\ldots$

Table 3.1: Rule set for fwd and single_level properties

The rule for the *step* operator, shows that the Stripe produced by a *step* operator retains the f property (meaning any of the fwd or single_level properties) only if its right input Stripe R satisfies such property. Similarly, for the *flt* operator, that retains f properties only if its left input Stripe L satisfies such properties. For the union set operator, property f is retained at the produced Stripe only if it holds for the Stripe produced by merging Stripes L and R (in essence for the union of the input Stripes). On the other hand, for the set intersection operation, if any input Stripe satisfies property f in isolation, it is enough to ensure that their intersection will also satisfy such property as well. For the set difference operator *except*, it is sufficient to know whether the left input Stripe satisfies property f. For any operator that accepts a single input Stripe, such as the *select* operator, it is sufficient to check whether the input Stripe satisfies f property. Finally, for the lowest-level operator in a logical evaluation plan, the *scan* operator, there are two possibilities: The first is that it accepts a single input Stripe. In such case, the single_level property always holds since by definition the operator accesses data from a single Stripe that contains all nodes of a certain label-path, having a common *level* value. This, combined with the fact that a scan operator always satisfies the doc_order property results in that it always satisfies the fwd property as well. When the *scan* operator accesses multiple Stripes though, the fwd property holds only when there is no Stripe such that its path is a prefix of any other of the rest of the input Stripes. If this holds, then it is guaranteed that the produced Stripe will not contain any nodes having an ancestor-descendant relationship:

$$\mathsf{fwd}(scan(S_1,\ldots,S_n)) = \forall S_i,S_j\ (\mathsf{path}(S_i) \notin \mathsf{prefix}(S_j)), \text{ with } i,j \in [1,n] \text{ and } i \neq j$$

Regarding the single_value property, this holds only when all input Stripes have the same *level* value:

$$\mathsf{single\_level}(scan(S_1,\ldots,S_n)) = \bigwedge_{i=1}^{n-1} \big(\mathsf{level}(S_i) = \mathsf{level}(S_{i+1})\big)$$

Knowledge of such properties during the optimisation phase can be beneficial for selecting more efficient evaluation algorithms. These are discussed in Section 3.4.

3.2.3 Plan Generation

The conceptual operators described above, are used to form logical query evaluation plans for XPath expressions. To that end, we use a one-to-one mapping from XPath syntax tree nodes to conceptual operators and build evaluation plans based on the XPath operator semantics. The logical evaluation plans are then used to provide query execution plans. We now describe the implementations of the most important operators with respect to to our approach: the *step*, *flt* and *scan* operators.

3.3 Evaluation Algorithms

Query execution plans are trees of physical operators. Tree edges connect operators; from the lowest level ones that access data (access methods) to the highest level (root) operator that produces the query result. Each operator's task is to consume the output of its input operator(s) and produce its own output according to the operator semantics. This is in turn treated as input to an upper level operator.

The way that operators communicate their results is of paramount importance to query evaluation. The simplest, yet most expensive method is to materialise the operator output and then access it upon request. This implies the extra cost of writing the operator output, also called intermediate or temporary result, plus the cost of reading it whenever required. To avoid or minimise intermediate result materialisation, we adopt the *iterator* execution model, as described in [46]. All operators implement the iterator interface according to which there are three basic calls:

open() Initialises the operator.

next(*hint*) Produces the next output item.

close() Performs the operator's housekeeping.

All calls in the execution plan are propagated downwards (root to leaf). Whenever an operator needs an item from a specific input operator, then the latter simply performs a next() call and produces that item. This effectively means that many operations can now be pipelined, thus eliminating the need for intermediate result materialisation that would increase their I/O cost.

Physical operators process XML nodes. Operators need to communicate results to each other. We have already described that each next() invocation of an operator produces an output node; the callee operator communicates its result to the caller operator. In addition, in certain circumstances, we need to direct the execution of an input operator. This is accomplished by passing a "hint" node as an argument from the caller operator to the callee input operator. The complete interface for an operator's next() method is $\mathsf{next} : Node \rightarrow Node$. The effect of the hint argument on the operator process, differs according to the operator's semantics. It is discussed later, as we present each operator individually.

3.3.1 Access Methods

This category of physical operators solely corresponds to those operators that access data. Although their implementation details are strongly coupled with the storage architecture (which will be presented in the following chapters), we provide a description of their functionality. As described in Section 2.2, the striped representation of an XML tree instructs that nodes are contained in Stripes; to gain access to XML nodes, one must be able to scan Stripes. This is the purpose of our access methods: to provide access to Stripes. We first describe a single Stripe Scan operator and then present a Merge Scan operator which merges the contents of multiple Stripes into one.

3.3.1.1 Stripe Scan

This is the scan operator for a single Stripe; it produces the nodes contained in the Stripe in document order. Each time the $\mathsf{next}(hint)$ method is called from a parent operator, a unique node n is returned satisfying predicate $n.start \geq hint.start$. This is essentially a contract between the caller and a scan operator that ensures that each invocation of $\mathsf{next}(hint)$ will always produce a node that is either *hint* or occurs *after hint*. Note that two successive calls of next() with the same value of *hint* argument, will never produce the same node; two distinct nodes n_1, n_2 will be returned for which the following will hold: $hint.start \leq n_1.start < n_2.start$.

Consider a Stripe Scan operator as a forward cursor (iterator) over a Stripe. At any given time, the cursor points to the current node, say *cur_node*. Calling next(), it is possible to retrieve the node that follows *cur_node* (in document order) or, by using an appropriate *hint* value, locate the first node n having $n.start \geq hint.start$ and thus skip all nodes between *cur_node* and n. A complete, sequential scan of a Stripe S runs in time proportional to its size *i.e.,* $O(|S|)$.

3.3.1.2 Merging Stripe Scans

It is often the case that multiple Stripes need to be scanned as one. When this is required, we use a Merge Scan operator. Its task, as the name suggests, is to merge the result of its input Stripe Scan operators and produce the merged output in document order. Note that no extra sorting operation is necessary before merging, as the input Stripe Scan operators produce nodes in document order.

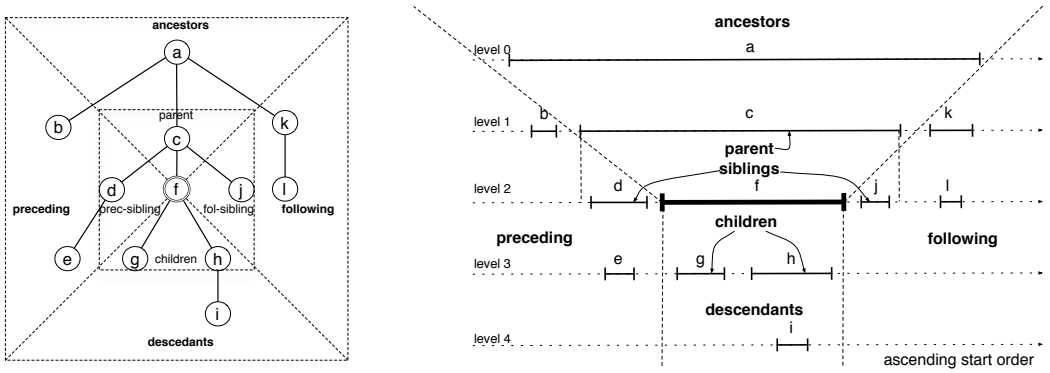
For each invocation of the next(*hint*) method, the Merge Scan operator produces a node n that (a) has the minimum *start* value among all nodes returned from the input Stripe Scan operators, and (b) satisfies the *hint* predicate: $n.start \geq hint.start$. For a complete scan of k Stripes, we need time proportional to the size of the Stripes *i.e.,* $O(\sum_{i=0}^{k}|S_i|)$. In addition, a Merge Scan operator needs extra time to identify the input operator that produced the node with the minimum *start* value. This can be accomplished in $O(\lg k)$ time using a priority queue and thus the required time for scanning and merging k Stripes is $O(\lg k \sum_{i=0}^{k}|S_i|)$.

3.3.2 Structural Joins

As described in Section 3.1, XPath is mainly used for locating XML tree nodes. The expression that performs such a task is the *axis step* expression, the so called *location step* (a::n). Ignoring the node test part (n) for now, which is mainly used for selecting nodes according to their type or label, it is the axis part a of a location step that specifies the direction of the step expression.

To that end, we provide a family of navigation algorithms which, starting from a context node sequence, allow the identification (selection) of tree nodes, with respect to a given axis. The most commonly used name for these operations is *structural joins* [8], as the selection of the produced tree nodes is accomplished through operations on the structural information of tree nodes.

We now provide the general properties of the structural join operator. To begin



(a) An XML tree

(b) The interval representation

Figure 3.2: Four major partitions defined for XML node f

with, it has exactly two input operators; the left input operator provides the context node sequence, while the right input operator provides a sequence of candidate nodes from which the output nodes will be selected, according to the operator semantics. Both input operators *must* produce nodes in document order, *i.e.,* satisfy the doc_order property; this is a prerequisite for our structural join operators, so no extra sorting of tree nodes or extra tests on their structural information are necessary. This means that two consecutive next() invocations $i, i+1$ of an input operator, will produce nodes n_i, n_{i+1} for which $n_i.start < n_{i+1}.start$ *i.e.,* $n_i \ll n_{i+1}$.

We have already described that each invocation of the next() method is accompanied with a *hint* argument in order to be able to direct the execution of an operator. For our structural join operators, the *hint* argument is only used for retrieving context nodes, *i.e.,* it is only used for the left input operator. This is only natural since for a structural join, the context node is the node of reference according to which the candidate nodes are finally selected.

It is important for the efficient implementation of structural join operators, to consider the tree structural properties of the context nodes along with the candidate nodes, a subset of which will be finally selected, according to the semantics of a location step. In [47], the authors discuss a set of *major axes*, namely the *descendant*, *ancestor*, *following* and *preceding* axis, that for any XML node define a partitioning of the XML tree. This partitioning is illustrated in Figure 3.2(a). Consider tree node f. The four major axes define four partitions of the document tree with respect to node f. The partitions (large triangles) *below*, *above*, *before* and *after* node f, enclose all descendant, ances-

tor, preceding and following nodes of f. Similarly, the small triangles that exist within the large triangles and thus major partitions of the XML tree, define sub-partitions for the parent, children and sibling nodes of f. This provides useful information regarding the part of the document that is relevant to the evaluation of a location step of axis a with respect to a context node. Similar observations can be made on the interval representation of a document tree, depicted in Figure 3.2(b). In the same spirit to the query axis windows defined in [47], we provide a mapping function that maps a context node u with a *candidate range* with respect to axis a. A candidate range for a context node u is a boundary interval range where all candidate nodes lie (their *start* value) with respect to axis a. The following candidate ranges are defined:

$$\begin{aligned}
\mathsf{range}(u, child) &= (u.start, u.end] \\
\mathsf{range}(u, parent) &= [u.par, u.par] \\
\mathsf{range}(u, descendant) &= (u.start, u.end] \\
\mathsf{range}(u, ancestor) &= (r.start, u.par] \\
\mathsf{range}(u, preceding) &= (r.start, u.start) \\
\mathsf{range}(u, following) &= (u.end, r.end) \\
\mathsf{range}(u, preceding\text{-}sibling) &= (u.par, u.start) \\
\mathsf{range}(u, following\text{-}sibling) &= (u.end, u.par_end)
\end{aligned}$$

where *par_end* is the *end* value of a node's parent node, while *r* stands for the document root node.

We now present the structural join operators for forward axes: *descendant*, *child*, *following* and *following-sibling*. For each axis considered for the location step evaluation (a::n), we first provide a description of its naïve evaluation for a sequence of context nodes; this involves the evaluation of the location step once per context node (the *active* context node) in the context node sequence. Each evaluation produces a node sequence. Then, the output node sequences for all context nodes are merged, removing duplicate nodes and finally produce a duplicate-free output node sequence, sorted in document order as defined in the XPath specification [14]. This description, along with determining the candidate range for a context node with respect to axis a, provides useful insight for potential inefficiency issues. One common problem is that the candidate ranges of two distinct context nodes overlap. When this happens, it is likely that the concatenation of the produced sequences for each context nodes, will not produce a node sequence containing nodes in document order. This can have a

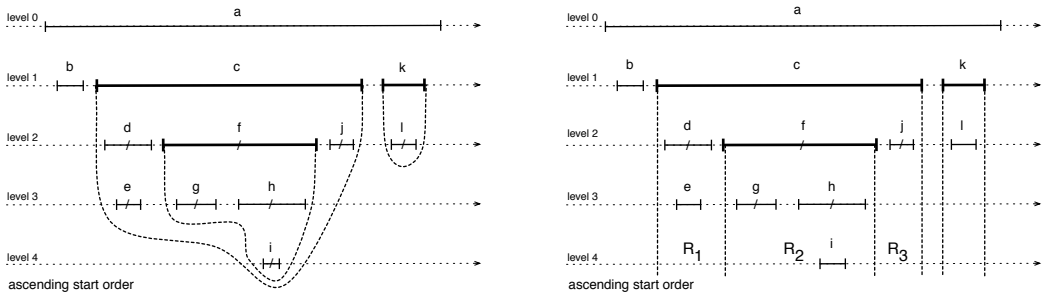
negative impact on the overall query evaluation time as it introduces the need for a sorting operator, which not only adds to the overall computational and I/O costs, but also blocks the query plan. In addition, overlapping candidate ranges for two distinct context nodes may result in node sequences having common XML nodes. When these sequences are finally merged, there is a need for a *distinct* (based on node identity) operator, which also implies a sorting operation. Finally, the overlapping candidate ranges indicate that some parts of a candidate node sequence must be examined multiple times in order to produce the result sequences for each context node. Our goal is to provide efficient structural join algorithms that deal with these issues.

3.3.2.1 Descendant Axis Structural Join

The naïve approach for the descendant axis structural join algorithm is to iterate over the input context node sequence and for each active context node, identify (the sequence of) its descendant nodes from the input candidate nodes. Then, the output node sequences for all context nodes must be merged, removing duplicate nodes and finally produce the sorted, duplicate-free output node sequence. For instance, consider the XML tree depicted in Figure 3.1(a) and the context node sequence: (c, f, k). The identification of the descendant nodes for nodes c, f and k yields node sequences (d, e, f, g, h, i, j), (g, h, i) and (l) accordingly. After merging the produced node sequences, sorting the result and removing duplicate nodes, the final output node sequence is: $(\overbrace{d, e, f, \underbrace{g, h, i}_{f}, j}^{c}, \overbrace{l}^{k})$.

Let us now consider what makes such an evaluation algorithm inefficient. To begin with, it produces duplicate nodes: Nodes g, h and i are produced for context nodes c and f. This means that a set of candidate nodes are repeatedly fetched. Most importantly though, these duplicates nodes must later be removed in order to provide the final, duplicate-free output node sequence. As already described, this operation yields a blocking operator since the intermediate (possibly large) result must be first stored and then sorted in order to remove duplicates and provide the final sequence sorted in document order.

How can we avoid scanning the same candidate nodes and produce duplicate nodes? Due to the tree structure of the XML document model, one can observe that the descendant nodes of a certain node are all its reachable nodes *i.e.,* its subtree. This effectively bounds the range that should be considered when searching for descendant nodes with



(a) Descendant axis (b) Child axis

Figure 3.3: Node relationships and structural joins (a)

respect to a context node c to $(c.start, c.end]$. Now let us consider two context nodes c_1 and c_2 with $c_1 \ll c_2$, meaning that $c_1.start < c_2.start$. There are two possibilities: Node c_2 is either a descendant or a following node of c_1. When c_2 follows c_1, their two subtrees are disjoint and thus there cannot exist a node that is a descendant of both c_1 and c_2. This is also easy to observe when considering their candidate ranges: When context node c_2 follows c_1, we have that:

$$\overbrace{c_1.start \leq c_1.end}^{c_1\text{'s range}} < \overbrace{c_2.start \leq c_2.end}^{c_2\text{'s range}}$$

However, when c_2 is a descendant of c_1 *i.e.,* node c_2 is member of c_1's subtree, then c_1's subtree contains c_2's subtree and thus duplicate nodes will be produced: all nodes that belong to c_2's subtree. This is also evident when considering their candidate ranges:

$$\overbrace{c_1.start < \underbrace{c_2.start \leq c_2.end}_{c_2\text{'s range}} \leq c_1.end}^{c_1\text{'s range}}$$

Consider the example in Figure 3.3(a). Context node k follows node c and thus their candidate ranges do not overlap This implies that the node sequences, produced for the descendant axis location step are disjoint. In fact, the concatenation of the produced sequences faithfully maintains document order: $(\overbrace{d,e,f,g,h,i,j}^{c}, \overbrace{l}^{k})$. The same holds for context nodes f and k. However, when considering context nodes c and f, where f is a descendant of c, the descendant nodes of f are also descendant nodes of c, resulting in duplicates when both context nodes are processed.

This observation leads to a very useful conclusion: Any context node that is a descendant of a context node that has already been processed, will not contribute any

Algorithm 3.1: desc_sj::next (*hint*)

```
1  begin
2      exit ← false;
3      repeat switch (state) do
4          case NEXT-CTX
5              hint.start ← max(hint.start, ctx.end + 1);
6              ctx ← left_input.next(hint);
7              if (ctx.start < cnd.start) then
8                  state ← CHECK-CND;
9                  break ;
10         case NEXT-CND
11             hint.start ← ctx.start + 1;
12             cnd ← right_input.next(hint);
13         case CHECK-CND
14             if (cnd.start ≤ ctx.end) then
15                 state ← NEXT-CND;
16                 exit ← true;
17             else
18                 state ← NEXT-CTX;
19             break ;
20     until (exit = true);
21     return cnd;
22 end
```

new output descendant nodes and thus can thus be skipped. If all such context nodes are skipped, no duplicate nodes will be produced since each of the remaining context nodes follows the ones already processed. Taking advantage of this, the main idea of the descendant axis structural join algorithm is to iterate over the input context node sequence, skipping all context nodes that produce duplicate nodes. Then, for each active context node, we merely identify its descendant nodes from the input candidate nodes. No duplicate nodes are produced while we output nodes in document order. The algorithm is presented in Algorithm 3.1. and is conceptually divided in three distinct operations: (a) a context node retrieval (from the left input operator) (b) an output candidate node retrieval (from the right input operator) and (c) a descendant axis predicate test between context and output candidate nodes. Within a single next() call, one or more of the operations mentioned above are repeatedly executed, until an output node is produced. A state variable is used for driving the execution flow as required. Whenever a context or candidate node is needed, the state flag is set to NEXT-CTX or NEXT-CND value accordingly. Similarly, for testing the axis predicate, the state flag must be set the CHECK-CND value.

A typical execution flow of the algorithm involves the following order: Suppose we have already retrieved a context node (*ctx*), the active context node. We now need to retrieve an output candidate node (*cnd*) that lies within the candidate range of the active context node in order to be its descendant, *i.e.,* $cnd.start \geq ctx.start + 1$ (lines 11-12). Should the retrieved candidate node satisfy the descendant axis predicate (line 14), we output the current candidate node and set the state flag to NEXT-CND value so that the next invocation of next() will continue by retrieving the next candidate node according to the active context node (lines 15-16). If the axis predicate fails, it is certain that we have exhausted all possible output nodes (if any) for the active context node and therefore we need to retrieve the next context node (line 18). During retrieval, we skip any descendant context nodes by requesting the next context node that *follows* the active one (lines 5-6). The process repeats until we have a match for the descendant axis predicate. Note that if any of input operators is exhausted, the process is completed returning a special EOF node. This is not described in Algorithm 3.1 for clarity.

3.3.2.2 Child Axis Structural Join

In contrast to the descendant axis structural join operator, the naïve approach for the child axis counterpart does not produce node sequences that contain duplicate nodes. This is due to the tree structure of the XML data model; each node has exactly one, unique parent node and thus it is impossible for the same node to exist in node sequences produced for different context nodes. Nevertheless, the naïve approach is still inefficient since the nodes contained in the produced node sequences for a series of context nodes may be interleaved; as a result a final sorting operation is still required for producing the final output node sequence in document order. Again, consider the XML tree depicted in Figure 3.1(a) and the context node sequence: (c, f, k). The identification of the children nodes for context nodes c, f and k yields node sequences (d, f, j), (g, h) and (l) accordingly. The final output node sequence is: $(\overbrace{d,f}^{c}, \overbrace{g,h}^{f}, \overbrace{j}^{c}, \overbrace{l}^{k})$.

Since no duplicates are generated, we turn our focus to producing output nodes in document order. Since the set of children nodes of a context node is a subset of its descendant nodes, we may bound the candidate range of a context node c to $(c.start, c.end]$. Let us again consider two context nodes c_1 and c_2 with $c_1 \ll c_2$, *i.e.,* c_2 is either a descendant or a following node of c_1. When c_2 follows c_1, their candidate ranges do not overlap and in particular all nodes in c_1's candidate range occur before

any node that belongs in c_2's candidate range since

$$\overbrace{c_1.start \leq c_1.end}^{c_1\text{'s range}} < \overbrace{c_2.start \leq c_2.end}^{c_2\text{'s range}}$$

This is shown in the example in Figure 3.3(b). Context node k follows node c and therefore the concatenation of their produced node sequences maintains document order: $(\overbrace{d, f, j}^{c}, \overbrace{l}^{k})$. Likewise, for context nodes f and k: $(\overbrace{g, h}^{f}, \overbrace{l}^{k})$. However, when c_2 is a descendant of c_1 *i.e.,* node c_2 lies within the candidate range of c_1:

$$\overbrace{c_1.start < \underbrace{c_2.start \leq c_2.end}_{c_2\text{'s range}} \leq c_1.end}^{c_1\text{'s range}}$$

and the nodes produced from the evaluation of child axis step *w.r.t.* both context nodes are interleaved. Observe context nodes c and f of Figure 3.3(b), with f being a descendant node of c. The concatenation of the produced node sequences produces a node sequence that does not contain nodes in document order.

As described above, what causes the output nodes to be produced in a non-sorted order is the existence of context nodes that are descendants of other context nodes. Inspired by the work in [49], we employ a range-partitioning technique which allows the output nodes be produced in document order: When we consider a single context node c, we scan its candidate range $(c.start, c.end]$ for its children nodes *i.e.,* we search for nodes having their *start* value within its candidate range. However, if it contains descendant context nodes then we need to keep a sequence of context nodes and partition their candidate ranges in order to provide an *active* context node for an *active* range at a time and produce a sorted (in document order) output node sequence. We use a stack to maintain the sequence of context nodes, the *context node stack*. The top of the stack provides the active context node at any time.

We now demonstrate the range partitioning technique. Consider the context node sequence (c, f, k). At first, we retrieve context node c and push it in the context node stack; node c is now the active context node. We then retrieve context node f and compare it to the active one. Since f is a descendant of c, it partitions c's range $(c.start, c.end]$ to three sub-ranges: R_1: $(c.start, f.start]$, R_2: $(f.start, f.end]$ and R_3: $(f.end, c.end]$. At first, we merely operate on the active sub-range R_1 and produce children nodes of c that occur *before* context node f (including f): nodes d and f are produced. We then push context node f in the stack and retrieve the next context node, k, which is now compared to the new active context node (f). Since k follows

f, there is no context node being a descendant of the active one, and it is thus safe to produce all its children nodes. Sub-range R_2 becomes active and nodes g and h are produced. We no longer need context node f and it is therefore popped out from the stack. Node c becomes active again and is now compared to the context node lastly retrieved, k. Context node k follows c and is now safe for the rest of c's children nodes to be produced, operating on sub-range R_3. Node j is now produced, context node c is popped out from the stack with context node k replacing it and the process continues in a similar manner. Observe that the final, produced node sequence (d, f, g, h, j, k) faithfully maintains document order.

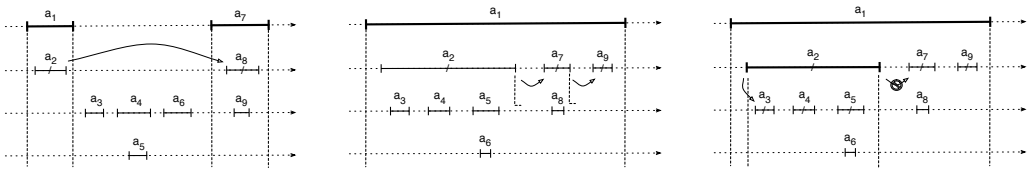
The algorithm for the child axis structural join operator (next() method) is outlined in Algorithm 3.2. The process is divided in the following basic operations: (a) context node retrieval (from the left input operator) (b) output candidate node retrieval (from the right input operator), and (c) the child axis predicate test between context and output candidate nodes.

The partitioning of the context node's candidate range and the active range selection occurs whenever a context node is requested (lines 4-14). This process uses stack manipulation operations such as pushing the current context node (ctx) onto the stack and retrieving the next one (lines 30-34), as well as popping the active context node from of the stack (lines 36-38). Regarding the context node range partitioning, the rationale is that we maintain a stack of context nodes, with each context node being a descendant of all nodes between that and the one at the stack's bottom. Suppose we have already retrieved a context node, the *current* context node (ctx). We first check the context node's stack status. If the stack is empty we simply push the current node to the stack and retrieve the next context node. Otherwise, we compare it to the active context node (the top of the stack) and if it is its descendant, we schedule to push it to the stack but only after we (schedule the) output of all children nodes of the active context node that exist before the current one. On the other hand, if the current follows the active context node, we schedule to pop it out of the stack but only after we (schedule the) output of all its children nodes. We continue in a similar manner comparing the current context node against the context node that is always active *i.e.,* the top of the stack.

For driving the execution flow of the algorithm as required but also being able to pipeline operations, we use a stack of states. The operation that will be executed is always determined by the state at the top of the state stack. To schedule an operation

Algorithm 3.2: child_sj::next (*hint*)

```
1  begin
2      repeat switch (state.top()) do
3          case NEXT-CTX
4              if (ctx_stack.empty()) then
5                  state.push(PUSH-CTX);
6                  break ;
7              else if (ctx.start ≤ ctx_stack.top().end) then
8                  state.push(PUSH-CTX);
9                  end ← ctx.start;
10                 state.push(FIRST-CND);
11             else
12                 state.push(POP-CTX);
13                 end ← ctx_stack.top().end;
14                 state.push(FIRST-CND);
15         case FIRST-CND
16             if (cnd.start > ctx_stack.top().start) then
17                 state.pop();
18                 state.push(CHECK-CND);
19                 break ;
20         case NEXT-CND
21             cnd_hint.start ← max(ctx_stack.top().start + 1, min(cnd.end + 1, end));
22             cnd ← right_input.next(cnd_hint);
23         case CHECK-CND
24             state.pop();
25             if (cnd.start ≤ end) then
26                 state.push(NEXT-CND);
27                 if (cnd.level = ctx_stack.top().level + 1) then exit ← true;
28             break ;
29         case PUSH-CTX
30             state.pop();
31             ctx_stack.push(ctx);
32             ctx ← left_input.next(hint);
33             if (ctx = EOS) then state.clear(); state.push(REST-CTX);
34             break ;
35         case POP-CTX
36             state.pop();
37             ctx_stack.pop();
38             break ;
39         case REST-CTX
40             if (¬ctx_stack.empty()) then
41                 state.push(POP-CTX);
42                 end ← ctx_stack.top().end;
43                 state.push(FIRST-CND);
44             else cnd ← EOS; exit ← true;
45             break ;
46     until (exit = true); return cnd;
47 end
```



(a) (b) (c)

Figure 3.4: Skipping unmatched candidate nodes

b to occur after operation a, we simply push first the state for operation b and then the state for operation a.

The retrieval of a candidate child node operation (lines 21-22) is responsible for the retrieval of children candidate nodes by calling the next() method of the right input operator. In order to skip unmatched candidate nodes, we make use of information that is extracted from the active context node combined with the active range and the candidate node that was produced last. There are two possible skipping opportunities regarding candidate nodes; the first is based on the ancestor-descendant relationship between a context node and their prospective children nodes while the second relies on the sibling relationship between children nodes of the same context node:

Descendant-based skipping As already mentioned, the range of a node c that is a child of a node p is included in its parent's range *i.e.,* $p.start < c.start \leq c.end \leq p.end$, since $\mathsf{children}(p) \subseteq \mathsf{children}^{+}(p)$. Therefore, at any given time, we are only interested in candidate nodes that lie within the active context node candidate range, *i.e.,* having a *start* value greater than the active context node's *start* value. This effectively skips all candidate nodes that exist *before* the active context node and thus would not contribute to the output result. Consider the example shown in Figure 3.4(a) and context node sequence (a_1, a_7). When context node a_1 is active, we merely produce node a_2. Context node a_7 follows a_1 and as such a_1 is discarded and a_7 becomes active. Using $a_7.start$ value as a hint for retrieving candidate nodes, we effectively skip all candidate nodes that occur before a_7 and thus do not satisfy the child axis predicate with respect to the active context node. Candidate node a_8 is then retrieved and produced.

Sibling-based skipping All children nodes of a certain node are siblings. Sibling nodes *follow* each other *i.e.,* if c_1, c_2 are siblings with $c_1 \ll c_2$, then $c_2.start > c_1.end$. Having already produced a child node for an active context node, we

exploit the sibling relationship that children nodes share in order to skip candidate nodes that are descendants of the context node but not its children nodes *i.e.,* skip $\mathsf{children}^+(p) - \mathsf{children}(p)$. This is depicted in Figure 3.4(b), where for active context node a_1 and after producing child node a_2, we use its *end* value as a hint for retrieving a candidate node. This results in skipping a_2's subtree *i.e.,* nodes $a_3 \ldots a_6$. This skipping technique has a single restriction: The suggested hint must be within the *active range*. For example, consider that node a_2 is also a context node and thus the context node sequence is (a_1, a_2) (Figure 3.4(c)). Node a_1 is pushed onto the stack and we now retrieve context node a_2. Since a_2 is a descendant of a_1, the active range is: $(a_1.start, a_2.start]$ and candidate node a_2 (now retrieved from the right input operator) is identified as a_1's child node. Should we use its *end* value as a hint to retrieve the next candidate node, we would skip all its descendant nodes and thus its children nodes that are later in need when a_2 will be the active context node. Instead, since $a_2.end$ is beyond the active range, we simply ignore it and retrieve the next candidate node: a_3. Context node a_2 then becomes active, the active range is: $(a_2.start, a_2.end]$ and candidate node a_3 is produced.

3.3.2.3 Following Axis Structural Join

Similar to its descendant axis counterpart, the naïve approach for the following axis structural join produces node sequences that contain duplicate nodes. Again, consider the XML tree depicted in Figure 3.1(a) and the context node sequence: (c, f, k). The identification of the following nodes for context nodes c, f and k yields node sequences (k, l), (j, k, l) and $()$ accordingly. The final output node sequence is: $(\underbrace{j, \overbrace{k, l}^{c}}_{f})$.

In fact, for a long sequence of context nodes, the number of duplicate nodes produced will be significantly large: For a node to follow a context node, it only needs to occur *after* the context node. This implies that the candidate range for the following axis with respect to a context node c is not bounded: a candidate following node of a context node c, lies within range $(c.end, r.end)$, where r is the document root node. This effectively means that we merely need to identify the context node with the minimum *end* value since any other context node c' (with $c'.end \geq c.end$) will produce a node sequence that any of its node members are also included in the sequence produced for c. For instance, the context sequence (c, f, k) can be replaced with the sequence (f) since context node f has the minimum *end* value and thus its produced sequence

Algorithm 3.3: fol_sj::next (*hint*)

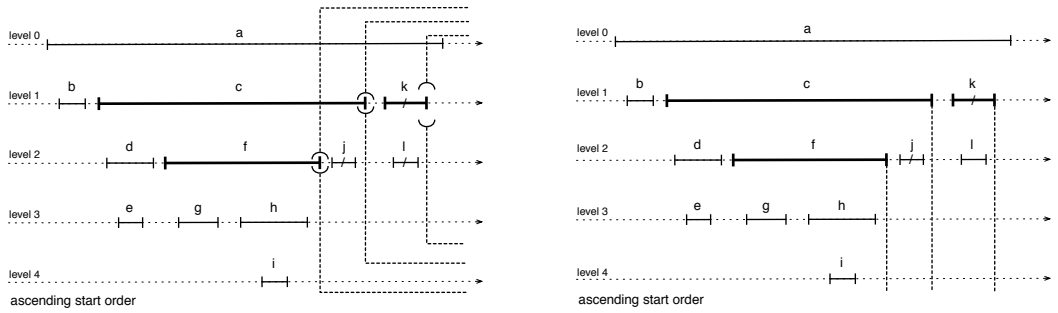
```
1  begin
2      repeat switch (state) do
3          case NEXT-CTX
4              ctx ← left_input.next(hint);
5              repeat
6                  tmp ← ctx;
7                  ctx ← left_input.next(hint);
8              until (ctx.end ≤ tmp.end) ;
9              ctx ← tmp;
10             state ← NEXT-CND;
11         case NEXT-CND
12             hint.start ← ctx.end + 1;
13             cnd ← right_input.next(hint);
14             exit ← true;
15             break ;
16     until (exit = true);
17     return cnd;
18 end
```

of following nodes (j,k,l) will include any member of the produced node sequences of any of the rest context nodes. This is shown in Figure 3.5(a). Note that the output node sequence is in document order and does not contain any duplicate nodes.

The algorithm is presented in Algorithm 3.3. and it is divided in two operations: (a) the context node identification (from the left input operator) and (b) an output candidate node retrieval (from the right input operator) The identification of the context node *only* occurs during the first invocation of the next() routine: Context nodes are repeatedly retrieved until the one having the minimum *end* value is determined (lines 4-9). Then, the first candidate node that lies within the candidate rage of the context node *i.e.,* $cnd.start > ctx.end$, is requested and returned (lines 12-14). Observe that there is no need for an axis predicate test; the following axis predicate is ensured by the input operator's semantics: If a candidate node is returned, it is ensured that it follows the context node. During subsequent invocations of the next() routine, we merely output the next retrieved candidate node.

3.3.2.4 Following-Sibling Axis Structural Join

The structural join operator for the following-sibling axis resembles the child axis structural join operator: they both involve the evaluation of a parent-child relationship.



(a) Following axis (b) Following-sibling axis

Figure 3.5: Node relationships and structural joins (b)

And although it seems straightforward for the child axis operator to evaluate parent-child relationships, this does not hold for the following-sibling structural join, at least not at first glance. The following-sibling axis predicate involves the conjunction of two distinct predicates: the (a) following and (b) sibling predicates; it the second conjunct predicate that encapsulates the evaluation of a parent-child relationship since sibling nodes share the same parent.

In contrast to its child axis counterpart though, the naïve approach for the following-sibling structural join operator may produce node sequences with duplicate nodes. This occurs with sibling context nodes. For example, consider three sibling nodes $s_1 \ll s_2 \ll s_3$ and a context node sequence containing the first two siblings: (s_1, s_2). Node s_3 will be produced in the output sequences for both context nodes and thus a distinct operator is needed. In addition, as in the case of the child axis operator, the nodes contained in the produced node sequences for a sequence of context nodes may be interleaved when the result sequences are concatenated and thus a sorting operation is required. To verify this, consider again the XML tree depicted in Figure 3.1(a) and the context node sequence: (c, f, k). The identification of the following nodes for context nodes c, f and k yields node sequences (k), (j) and $()$ accordingly. The final output node sequence however is: $(\overbrace{j}^{f}, \overbrace{k}^{c})$.

Let us examine in detail the identification of following-sibling nodes of a context node c. As mentioned above, the following-sibling axis involves two predicates: the following predicate, which enforces that a candidate node must occur *after* the context node, and the sibling predicate, which enforces that a candidate node cannot occur after context node's *parent* node. As a result, we have that the following-sibling nodes of a context node c (if any) must lie within candidate range $(c.end, c.par\_end)$.

Let us now consider context nodes c_1 and c_2 with $c_1 \ll c_2$ and examine the relationship between their candidate ranges. Again, we have that c_2 is either a descendant or a following node of c_1:

1. When context node c_2 is a descendant of c_1, we have that:

$$c_1.start < c_2.start \leq \overbrace{c_2.end \leq c_2.par\_end}^{c_2\text{'s range}} \leq \overbrace{c_1.end \leq c_1.par\_end}^{c_1\text{'s range}}$$

   which effectively means that all nodes of the following-sibling produced sequence with respect to c_2 precede any of the nodes of the produced node sequence with respect to context node c_1. Therefore the concatenation of the node sequences produced for context nodes c_2, c_1 (*i.e.,* in reverse order as produced for the context nodes) is adequate to produce the sorted, duplicate-free result node sequence.

2. The case where context node c_2 follows c_1 is of special interest. In order to study this in detail, we break it down into the following sub-cases:

   (a) c_2 is a sibling of c_1: In essence, we have that context node c_2 is itself a following-sibling node of c_1. As stated above, this is the case that duplicate nodes are produced when combining the produced node sequences. Regarding the context node candidate ranges, we have that:

$$c_1.start \leq \overbrace{c_1.end < c_2.start \leq \underbrace{c_2.end \leq c_1.par\_end}_{c_2\text{'s range}}}^{c_1\text{'s range}}$$

   where $c_1.par\_end = c_2.par\_end$ since c_1 and c_2 are siblings. This, in addition to the fact that both context nodes are siblings, results in a node sequence for c_2 that is included in the sequence produced for context node c_1. As a result, we may simply ignore the evaluation of context node c_2.

   (b) c_2 is a descendant of a sibling of c_1: This is the case where the produced sequences of following-sibling nodes for the two context nodes may be interleaved. This is due to the fact that c_2's candidate range is a sub-range of the one of c_1:

$$c_1.start \leq \overbrace{c_1.end < c_2.start \leq \underbrace{c_2.end \leq c_2.par\_end}_{c_2\text{'s range}} \leq c_1.par\_end}^{c_1\text{'s range}}$$

As an example, consider Figure 3.6 and context node sequence (a_2, a_6) where node a_6 is a descendant of a_1's sibling: node a_5. The produced nodes sequences for context nodes a_2, a_6 are (a_5, a_9) and (a_7, a_8) respectively while the resulting node sequence for the context node sequence is $(\overbrace{a_5}^{a_2}, \overbrace{a_7, a_8}^{a_6}, \overbrace{a_9}^{a_2})$.

(c) c_2 follows c_1 but it is neither c_1's sibling nor descendant of its siblings *i.e.,* it does *not* belong to c_1 parent's subtree: This is the case where the two subtrees rooted on the parent nodes of the context nodes are disjoint and in particular all nodes in c_1's parent subtree occur before any node that belongs to c_2's parent subtree. As a result, any node included in the produced sequence of following-sibling nodes for context node c_1 and thus in c_1's parent subtree, precedes any of the nodes that is included in the produced node sequence for c_2, as shown from the candidate ranges for the context nodes:

$$c_1.start \leq \overbrace{c_1.end \leq c_1.par\_end}^{c_1\text{'s range}} < c_2.start \leq \overbrace{c_2.end \leq c_2.par\_end}^{c_2\text{'s range}}$$

This is also shown in the example of Figure 3.6 when considering the context node sequence (a_6, a_{10}). The produced nodes sequences are (a_7, a_8) and (a_{11}) for context nodes a_6 and a_{10} respectively, while the resulting node sequence is produced by merely concatenating them in the same order, as these are produced for the context nodes: $(\overbrace{a_7, a_8}^{a_6}, \overbrace{a_{11}}^{a_{10}})$.

To summarise, in cases (1) and (2c) the candidate ranges do not overlap while in (2a) and (2b) one range is included in the other, partitioning it into sub-ranges. In order to support all cases in an efficient way, producing output nodes in document order, we adopt a range-partitioning technique as in the case of the child axis structural join operator. If we regard the above cases with respect to the parent node of the first (in document order) context node ($c_1.parent$), we have the following classification: Context node c_2 is either (a) a descendant node of $c_1.parent$ (cases (1), (2a) and (2b)) or (b) a following node of $c_1.parent$ (case (2c)). When context node c_2 is a descendant of c_1's parent node, it partitions c_1's candidate range $R_{c_1} : (c_1.end, c_1.par\_end)$[2] and therefore, we need to be able to *postpone* its evaluation for a range that is guaranteed that it will not produce any output nodes (for c_1), while later *resume* its evaluation for

[2]For case (1), c_2 can be considered that it partitions c_1's range to sub-ranges $\perp$ and R_{c_1}, where $\perp$ is the empty range.

specific range(s) that may produce output following-sibling nodes. To support that, we maintain a stack of context nodes; the top of the stack always provides the active context node while the rest of the context nodes in the stack are those, whose evaluation is temporarily suspended.

The algorithm for the following-sibling axis structural join operator (next() method) is outlined in Algorithm 3.4. The backbone of the algorithm is the same as the child axis structural join algorithm. The most significant changes are the invariant for the context node stack management along with the axis predicate handling. The process is again divided in the following basic operations: (a) context node retrieval (from the left input operator) (b) output candidate node retrieval (from the right input operator) and (c) the following-sibling axis predicate test between context and output candidate nodes.

The partitioning of the context node's candidate range and the "active" range selection occurs during the retrieval of the next context node (lines 4-17). As explained above, the rationale is that we maintain a stack of "suspended" active context nodes, with the top of the stack being the active context node currently evaluated. As in the child axis join operator, whenever the current context node is pushed onto the stack (becomes active), the next context node is retrieved becoming the new current one (line 32). Likewise, discarding the active context node means a pop operation from the stack and thus a new active context node (line 34). Now, suppose we have already retrieved a context node, the *current* context node (ctx). The current context node is then compared to the active context node (in fact, its parent node): If the current context node is a descendant of the active context node's parent (line 7), we evaluate the active context node for a sub-range that does not exceed the current context node (line 13). As soon as the sub-range is exhausted, the evaluation of the active context node is suspended and the current context node is pushed onto the stack (line 12). However, in the special case that the current and active context nodes are siblings (line 8), the active context node is evaluated within the sub-range $(active.end, current.start]$ (including the start of the current context node) (line 10), and is then discarded from the stack (line 9), with its sibling taking its place. This effectively means that in the presence of sibling context nodes, the one that encountered first is being evaluated until its sibling (context) node is reached so that no duplicate nodes are generated. In the case that the current context node follows the active context node's parent, we simply continue the evaluation of the active context node until the end of its candidate range and then we

Algorithm 3.4: fol_sibl_sj::next (*hint*)

```
1  begin
2      repeat switch (state.top()) do
3          case NEXT-CTX
4              if (ctx_stack.empty()) then
5                  state.push(PUSH-CTX);
6                  break ;
7              else if (ctx.start ≤ ctx_stack.top().par_end) then
8                  if (ctx.par = ctx_stack.top().par) then
9                      state.push(POP-CTX);
10                     end ← ctx.start + 1;
11                 else
12                     state.push(PUSH-CTX);
13                     end ← ctx.start;
14             else
15                 state.push(POP-CTX);
16                 end ← ctx_stack.top().par_end;
17             state.push(FIRST-CND);
18         case FIRST-CND
19             if (cnd.start > ctx_stack.top().end) then
20                 state.pop();
21                 state.push(CHECK-CND );
22                 break ;
23         case NEXT-CND
24             cnd_hint.start ← min(end, max(cnd.end + 1, ctx_stack.top().end + 1));
25             cnd ← right_input.next(cnd_hint);
26         case CHECK-CND
27             state.pop();
28             if (cnd.start < end) then
29                 state.push(NEXT-CND);
30                 if (cnd.par = ctx_stack.top().par) then exit ← true;
31             break ;
32         case PUSH-CTX
33             ...
34         case POP-CTX
35             ...
36         case REST-CTX
37             if (¬ctx_stack.empty()) then
38                 state.push(POP-CTX);
39                 end ← ctx_stack.top().par_end;
40                 state.push(FIRST-CND);
41             else cnd ← EOS; exit ← true;
42             break ;
43     until (exit = true); return cnd;
44 end
```

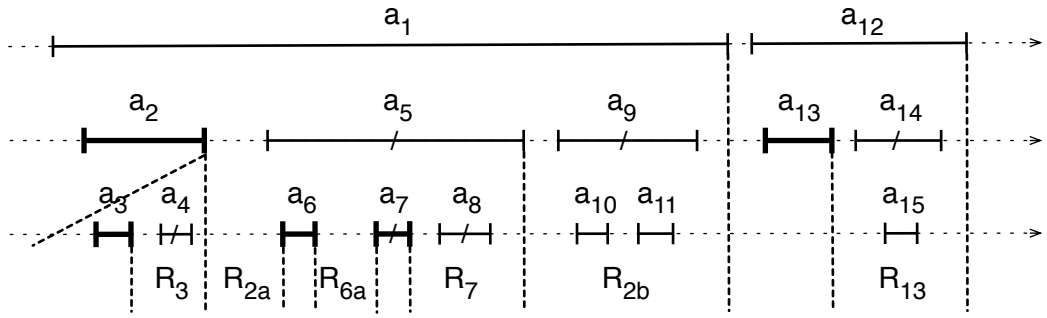


Figure 3.6: A more complex example for following-sibling axis structural join

discard it as it is no longer needed (lines 15 - 16). The process continues in a similar manner.

We now provide a detailed example of the following-sibling structural join execution. Consider the context node sequence $(a_2, a_3, a_6, a_7, a_{13})$ for the document tree (its interval representation) depicted in Figure 3.6. Suppose, we have already accessed the first context node *i.e.,* the current node is node a_2. During the first execution of next(), the context node stack is empty and therefore a_2 becomes active and node a_3 the new current context node. Note here that context node a_3 is a descendant of a_2 (case 1) and thus a descendant of a_2's parent node. Hence, we output candidate nodes for a_2 (active) until we reach $a_3.start$ and then push the current context node onto the stack. However, as a_3 is a descendant of a_2 and thus $a_3.start < a_2.end$, the active range for active context node a_2 is currently the empty range $\perp$. As a result, the evaluation of active context node a_2 is postponed. Node a_3 is now the active context node while a_6 is the current one. As context node a_6 follows a_3's parent node (case 2c), we evaluate node a_3 within range $R_3 : (a_3.end, a_2.end)$ and node a_4 is produced. As soon as we are done with the evaluation of active context node a_3 within range R_3, it is popped out from the stack. Now a_2 becomes active again and we can resume with its evaluation. However, the current node (a_6) is a descendant of its parent node (case 2b) and thus partitions its candidate range. The evaluation of the active context node will continue for active range $R_{2a} : (a_2.end, a_6.start)$ and candidate node a_5 will be produced. Current context node a_6 is then pushed onto the stack and becomes active, while a_7 is retrieved (current). Note that the active and current context nodes are siblings (case 2a). Active context node a_6 will be evaluated for range $R_{6a} : (a_6.end, a_7.start]$ and then popped out from the stack as any candidate node that lies in range $R_6 - R_{6a} = R_7$ will be produced

during the evaluation of context node a_7. Candidate node a_7 is produced for active context node a_6. Context node a_2 becomes active again (case 2b) but no following-sibling candidate node is produced until the end of its active range: $a_7.start$. Current context node a_7 is now pushed into the stack and a new context node is retrieved (a_{13}). Current context node a_{13} follows a_7's parent (case 2c), and thus we continue with the evaluation of context node a_7 for active range $R_7 : (a_7.end, a_5.end)$, producing candidate node a_8. a_7 is then popped out and a_2 becomes once again the active context node (case 2c), resuming its evaluation for final sub-range $R_{2b} : (a_5.end, a_1.start)$. Candidate node a_9 is produced, context node a_2 is finally discarded and context node a_{13} is pushed onto the (empty) stack. The evaluation continues likewise for the last context node of the node sequence and candidate node a_{14} is produced. The produced node sequence is $(a_4, a_5, a_7, a_8, a_9, a_{14})$.

Regarding the retrieval of a candidate following-sibling node operation (lines 24-25), we exploit the relationship between a context node and its candidate following-sibling nodes as well as the sibling relationship between the nodes of the produced node sequence (with respect to a single context node), in order to skip candidate nodes that will not contribute to the produced node sequence. As in the case of the child axis structural join algorithm, we make use of information that is extracted from the active context node combined with the active range and the candidate node that was lastly produced. At any given time, we seek for a candidate node that occurs *after* the active context node. In addition, if a candidate node has already been produced, we seek the next following-sibling node of the active context node *after* the one lastly produced. This last skipping rule must not be used when the current context node is within the active range of the active context node since that would skip the candidate nodes of the current context node. This set of rules for the candidate node retrieval is defined in line 24.

Finally, the following-sibling predicate test (lines 26-31), is performed between the active context node and the candidate node lastly retrieved. Any node that lies within the active context node's active range, follows the active context node and therefore is a following-sibling candidate. A candidate is produced only if it also satisfies the sibling predicate. As long as a candidate lies within the active range, we continue with the candidate node retrieval.

3.3.2.5 Alternative Following-Sibling Axis Structural Join

We now present an alternative algorithm for the evaluation of the following-sibling structural join. As described in Section 3.3.2.4, all following-siblings nodes of a certain context node c lie within its candidate range: $(c.end, c.par\_end)$. This demonstrates that for the efficient evaluation of the following-sibling axis structural join operation each node (either context or candidate node) must also be supplied with the full structural information of its parent node.

We remind the reader that in Algorithm 3.4, we consider the relationship between a current context node and the active context node's parent node in order to conclude whether the active context node's candidate range is partitioned and thus should be kept in the stack for future evaluation. We highlight though, that there are in fact two distinct cases that an active context node is kept in the stack for future evaluation: (a) when the current context node is a descendant of the active node (case 1) and (b) when the current context node is a descendant of the active's sibling (case 2b). We observe that a common point of reference in both cases is the fact that the current context node has a greater $level$ value than the active context node. Indeed, by checking the context node stack at any given time throughout the example described in Section 3.3.2.4, it is easy to observe that each time a context node is pushed onto the stack, it always has a greater $level$ value than the node at the top of the stack.

This is only natural though, considering the tree structure of the XML data model, in addition to the fact that active $\ll$ current always holds. To prove our claim, let us first consider the case that both active and current context nodes share the same $level$ value. This spans in two sub-cases: When the active and current context nodes are siblings, as already described, we merely produce candidate nodes that lie *before* the current context node (including current). Then, the active context node is discarded so that no duplicate nodes are produced. In the case that the current is not a sibling of the active context node, their candidate ranges do not overlap and thus as soon as the evaluation of the active context node is complete, it is safely discarded. The same holds in the case that the current context node has a smaller $level$ value than the active one: there is no overlapping of their candidate ranges and the active context node is discarded as soon as it is processed.

Generalising this observation, we present an alternative algorithm for the evaluation of the following-sibling structural join that can be used in the absence of the full structural information of parent nodes. The main idea remains the same: we partition

the candidate range of each context node into sub-ranges so that we are able to *suspend* its evaluation for a range that is guaranteed not to produce any output nodes, while later *resume* its evaluation for specific range(s) that may produce output following-sibling nodes. To support that, we maintain a stack of context nodes, according to the following invariant: *A context node is pushed onto the stack only if the stack is empty or it has a greater level value than the context node being at the top of the stack.* This invariant ensures that at any given time the "active" context node (top of the stack) is the one with the greatest *level* value among the context nodes, whose evaluation is temporarily suspended.

The algorithm is outlined in Algorithm 3.5. What differs from the original following-sibling structural join (Algorithm 3.4) is the range partitions generated for each active context node and the context node's stack management. The partitioning of the context node's candidate range occurs during the retrieval of the next context node (lines 3-13). As explained above, the rationale is that we maintain a stack of context nodes, with each context node having a greater *level* value than all nodes between that and the one at the stack's bottom. Note, that in the absence of (full) structural information of a node's parent, it is impossible to have knowledge of the exact candidate range of a context node. To that end, each context node (active) is processed within a range restricted by the *start* value of the next context node retrieved (current). Suppose we have already retrieved a context node, the *current* context node (ctx). The current context node is then compared to the active context node (top of the stack). If the current context node has a greater *level* value than the active one (line 5), we process the active context node until the current context node's *start* value is encountered (*i.e.,* within the range $(active.end, current.start)$) and it remains in the stack for future evaluation (lines 7 - 8). The current context node is then pushed onto the stack so that it becomes the active one (line 6) and a new current node is retrieved. This effectively covers cases (1) and (2b), where current context node is a descendant of the active context node or of any of its siblings. On the contrary, if the current context node's *level* value is less than or equal to the *level* value of the active context node (line 9), we process the active context node (within range $(active.end, current.start]$) and then discard it (lines 10 - 12) as (a) it is certain that it will not produce any other following-sibling nodes (case 2c) or (b) we want to prevent it from producing the rest of its following-sibling nodes (case 2a) to prevent duplicate generation.

As described above, the selection of the active range for a context node $c_1 : (c_1.end,$

Algorithm 3.5: alt_fol_sibl_sj::next (*hint*)

```
1  begin
2      repeat switch (state.top()) do
3          case NEXT-CTX
4              if (ctx_stack.empty()) then state.push(PUSH-CTX);
5              else if (ctx.level > ctx_stack.top().level) then
6                  state.push(PUSH-CTX);
7                  end ← ctx.start;
8                  state.push(FIRST-CND);
9              else
10                 state.push(POP-CTX);
11                 end ← ctx.start + 1;
12                 state.push(FIRST-CND);
13             break ;
14         case FIRST-CND
15             if (cnd.start > ctx_stack.top().end) then
16                 state.pop();
17                 state.push(CHECK-CND );
18                 break ;
19         case NEXT-CND
20             cnd_hint.start ← min(end, max(cnd.end + 1, ctx_stack.top().end + 1));
21             cnd ← right_input.next(cnd_hint);
22         case CHECK-CND
23             state.pop();
24             if (cnd.start < end) then
25                 if (cnd.par = ctx_stack.top().par) then
26                     state.push(NEXT-CND);
27                     exit ← true;
28                 else if (cnd.level > ctx_stack.top().level) then state.push(NEXT-CND);
29             break ;
30         case PUSH-CTX
31             state.pop(); ctx_stack.push(ctx);
32             ctx ← left_input.next(hint);
33             if (ctx = EOS) then
34                 end ← ∞;
35                 state.clear(); state.push(REST-CTX);
36             break ;
37         case POP-CTX
38             ...
39         case REST-CTX
40             if (¬ctx_stack.empty()) then
41                 state.push(POP-CTX);
42                 state.push(FIRST-CND);
43             else cnd ← EOS; exit ← true;
44             break ;
45     until (exit = true); return cnd;
46 end
```

$c_2.start$) is based on information obtained from a context node c_2 that is retrieved *after* c_1. This, depending on the input context node sequence, may lead to poor performance; the ending point of the selected active range for a context node may be significantly far from the ending point that would be selected if the full structural information of the active context node's parent was available ($c_1.par\_end$). This implies that more candidate nodes would be retrieved although they will not be produced in the output result. Most importantly, this may lead to an erroneous result; some of the nodes retrieved for an active context node and discarded because they do not satisfy the following-sibling axis predicate (with respect to the active context node), could be included in the result output of a context node that still remains in the stack, waiting to resume its evaluation for a sub-range that lies *after* ($c_1.par\_end$). In order to prohibit such erroneous behaviour, we include an extra condition that causes the evaluation of the active context node to halt, even if the end of its active range is not yet reached. We extend the tree-aware logic to the candidate nodes that are retrieved and compared against the active context node: When we begin the evaluation for the active context node, we retrieve a candidate node within the active range. If the candidate node does satisfy the following-sibling axis predicate and thus has a match, we output the retrieved node and schedule the process to continue with the retrieval of the next candidate node. If however, the candidate node does satisfy the following-sibling axis predicate, we will only continue with the retrieval of another candidate node if the discarded candidate node has a $level$ value greater than the value of the active context node. If this condition does not hold, then we can be certain that the active context node does not have any remaining following-sibling nodes as we have retrieved a candidate node that lies after the active context node's parent region. It is therefore safe to cease its evaluation and test the retrieved candidate node with the context nodes awaiting in the stack. Testing a candidate node that is not included in the result against the active context node acts as a safety net for the correct evaluation of the algorithm while at the same time optimises the overall process by preventing unnecessary retrieval of candidate nodes that will not contribute to the final result.

We now describe the evaluation of the new following-sibling structural join algorithm for the example tree in Figure 3.6 and for the context node sequence $(a_2, a_3, a_6, a_7, a_{13})$. The first current context node, node a_2, is pushed onto the empty stack and the new current context node is now node a_3. Context node a_3 is a descendant of a_2 (case 1) and thus since $a_3.level > a_2.level$. As a result, we process candidate nodes for active context node a_2 within the empty range $\perp$ (since $a_3.start < a_2.end$). We then push

the current context node a_3 onto the stack, while a_6 becomes the current context node. For context nodes a_3 and a_6 (case 2c), that share the same *level* value, we begin the evaluation for context node a_3 within range $(a_3.end, a_6.start]$. First, candidate node a_4 is produced as a following-sibling node of a_3. The next candidate node retrieved is node a_5 (which is a following-sibling node of the context node a_2 that is queued in the stack). Node a_5 is within the active range for the active context node a_3 but does not satisfy the axis predicate. However, since $a_5.level < a_3.level$ we stop the evaluation for context node a_3 as we know that there is no other candidate node that will satisfy the axis predicate and discard it. Now a_2 becomes again the active context node and we can resume with its evaluation for active range $(a_2.end, a_6.start)$ (case 2b), producing candidate node a_5. As soon as the evaluation is complete (the end of the range is reached), current context node a_6 is pushed onto the stack and becomes active, while a_7 is retrieved (current). Note that the active and current context nodes are siblings (case 2a) and therefore have the same *level* value. Active context node a_6 is evaluated for range $(a_6.end, a_7.start]$, producing candidate node a_7, and then popped out from the stack. Context node a_2 becomes active again (case 2b) but no following-sibling candidate node is produced until the end of its active range: $a_7.start$. Current context node a_7 is then pushed onto the stack as it has a greater *level* value than a_2. Current context node is now node a_{13} (case 2c), and since $a_{13}.level < a_7.level$, we continue with the evaluation of context node a_7 for active range $R_7 : (a_7.end, a_{13}.start]$, producing candidate node a_8. Then candidate node a_9 is retrieved, causing the end of the evaluation of context node a_7 ($a_9.level < a_7.level$). Context node a_7 is then discarded and a_2 becomes once again the active context node (case 2c), resuming its evaluation for active range $(a_5.end, a_{13}.start]$. Candidate node a_9 is produced, context node a_2 is finally discarded and context node a_{13} is pushed onto the (empty) stack. The evaluation continues likewise for the last context node of the node sequence and candidate node a_{14} is produced. The produced node sequence is $(a_4, a_5, a_7, a_8, a_9, a_{14})$.

3.3.3 Filter Processing

As described in Section 3.1, a *location step* expression is used for navigating the XML tree structure. In many cases, it is crucial to restrict the result of a location step; this is accomplished with a predicate list that follows the location step and filters its produced node sequence. We now focus on the evaluation of predicate expressions of the form: $E_1[E_2]$, where E_1 is an XPath expression that provides the input sequence

which will be filtered based on the predicate expression E_2. For each item in the input sequence, the result of the predicate expression is cast to a boolean value, the *predicate truth value*; the items of the input sequence for which the predicate truth value is false, are discarded. The casting of a predicate expression to a boolean value is called the *effective boolean value* of the predicate expression. In particular, when predicate expression E_2 results in a node sequence, its effective boolean value is true, only when the produced node sequence is not empty [14]. This section describes the evaluation of predicates of an XPath query. We call this *filter processing*.

It is common to describe an XPath query using a tree structure, a *twig pattern* [18]. For example, consider a simple XPath query A/B//C/D where all sub-expressions are location steps and "/", "//" are abbreviations for the child and descendant axis steps respectively. The semantics of such a query are captured by twig pattern: A-B=C-D where "-", "=" describe parent-child and ancestor-descendant relationships respectively. Predicate expressions in an XPath query introduce query branches; path expression A/B[.//C]/D is described using twig pattern: A-B[=C]-D, where node B of the query pattern is a branching node. Note that twig patterns merely provide the structural relationship among query nodes. To that end, other tree structures, have been proposed that capture in more detail the semantics of XQuery and thus XPath expressions (*e.g.,* [26, 83]).

Most existing works on twig query processing focus on returning the entire twig results, *i.e.,* tuples that contain matching nodes for *all* query pattern nodes [18]. In practice, however, returning the entire twig results is neither necessary nor efficient for XPath queries. Consider for example predicate expressions; the results for such expressions are never required in the final query result. In addition, in many cases, extra sorting and/or duplicate elimination operations have to be performed on the twig results in order to comply with query semantics. Another important issue considering twig pattern query processing is that most proposed methods (with the exception of [67], see Section 3.6) solely involve parent-child and ancestor-descendant tree node relationships, ignoring important constructs of XPath expressions such as other kind of tree node relationships (*e.g.,* the sibling relationship).

To evaluate predicate expressions along with navigational operations in a pipelined manner and be able to support any structural relationship between XML tree nodes, we adopt a general, iterative evaluation method, having the following characteristics:

- Predicate expressions are not fully evaluated, *i.e.,* we do not produce the complete result of a predicate expression but merely the first result with respect to a

given context node.

- There is no need for extra sorting operations since the (ordered) input sequence of context nodes is filtered on the fly and duplicate nodes are not produced.

- It supports the evaluation of preceding-following and sibling relationships in addition to parent-child and ancestor-descendant relationships. In general, it supports any of the XPath expressions that can be evaluated outside of predicates.

- It is fully pipelined, producing minimal result for the predicate expressions that is immediately discarded.

3.3.3.1 Filter Operator

The basic operator for supporting predicate expressions is a binary operator, the *Filter*. A Filter operator has two input operators; the left input operator provides the input context node sequence while the right input operator provides a (partial) node sequence which is produced for each context node and corresponds to the result sequence from the evaluation of the predicate expression with respect to the context node. The Filter operator then determines the effective boolean value of the sequence that is produced by the right input operator according to which the context node is either added to the result or discarded.

Note that the evaluation of the predicate expression is delegated to the query sub-plan rooted on the right input of the filter operator. The result node sequence of this sub-plan is the complete result for the predicate expression. However, for determining the effective boolean value of a node sequence, produced for a certain context node, we merely require to know whether the result sequence is empty or not in contrast to the computation of the complete result sequence. Consider the following XPath expression: A[B/C/D]. Each *a* node matching element tag A, may have many *d* descendant nodes of element tag D at path A/B/C/D. However, for the evaluation of the predicate expression, we do not need to produce the complete result for expression A/B/C/D. Instead, we can decide whether an *a* node will be kept in the result sequence as soon as we know that there exists at least one result node *d*, that is a descendant of *a* and matches the expression A/B/C/D. This may lead to significant computational and I/O savings when the complete result of a predicate expression is large in comparison to the filter input node sequence. We therefore turn our interest to quickly producing the

Algorithm 3.6: filter::next (*hint*)

```
1  begin
2      repeat
3          e1 ← left_input.next(hint);
4          if (e1 = EOF) then exit ← true;
5          else
6              e2 ← right_input.next(e1);
7              if (e2 <> ε) then exit ← true;
8      until (exit = true) ;
9      return e1;
10 end
```

first node of the complete result node sequence when evaluating a predicate expression for each node in the context node sequence.

This iterative, partial-evaluation approach for handling predicate expressions in XPath queries, requires new, specialised operators that follow the logic of filter processing. When an operator is used for evaluating a predicate expression we say that it operates in *filter mode*. An operator in filter mode, evaluates a specific sub-expression of the predicate expression. In order to do so, it performs a specific task with respect to a *reference* node *ref*, passed through the caller operator via method next(*ref*), which acts as the context node for the evaluation of the specific sub-expression. This will be explained in detail in the following sections.

3.3.3.2 Location Steps in Filters

This is the operator for evaluating a location step a::n in filter mode. Each invocation of next(*ref*) produces a node that belongs to the result sequence of expression $ref/a::n$, *i.e.,* reference node *ref* acts as the context node for the location path evaluation. Producing a node as a result implies that it satisfies the axis predicate a with respect to context node *ref*. The evaluation of location steps in filter mode are pushed down to the Stripe Scan operators. The context node is passed through argument *ref* of next() method and the scan operators return a result according the location step semantics. To that end, we employ the *Axis Stripe Scan* operator, when there is a single input Stripe and the *Axis Merge Scan* operator for handling multiple input Stripes.

Axis Stripe Scan This is the Stripe Scan operator, operating in filter mode. Its task, in addition to the normal Stripe Scan operator that merely retrieves nodes that are contained in a Stripe, is to retrieve nodes satisfying an axis predicate a with respect to

the context node *ref*, passed as an argument from the caller operator through next(*ref*). If none nodes exist satisfying the axis predicate for the context node, then next() returns an empty node ε to the caller operator.

The retrieval of the nodes contained in the Stripe occurs as follows: Suppose a next(ref_1) call occurs for a child Stripe Scan operator for some Stripe *S*. The first invocation will return node c_1 that is the first child node of context node ref_1 *that lies in Stripe S*. If node ref_1 has no children nodes (at Stripe *S*), then the empty node ε is returned. Subsequent calls of next(ref_1) will return the rest of the children nodes of ref_1 (stored at Stripe *S*), in document order. If, however, at any time the caller operator changes the context node argument, *e.g.,* calls next(ref_2), then the Scan operator will produce the first child node (if any) that is stored at Stripe *S* with respect to context node ref_2, ignoring the rest of the children nodes of ref_1, if not yet retrieved.

When next(*ref*) is called for the first time for context node *ref*, the context node's structural information is used in order to locate the first node (in document order) satisfying the axis predicate. For the descendant or child axes, we begin searching for a node that occurs after *ref.start* and return the first satisfying the descendant or child predicate respectively. Similarly, for the following or following-sibling axes, we begin searching for a node that occurs after *ref.end*. For any subsequent calls regarding the same context node, the scan operator will simply return the next available node that satisfies the axis predicate; otherwise the empty node ε.

An Axis Stripe Scan operator behaves as a forward cursor (iterator) over a Stripe. At any given time, the cursor points to a current node, say *cur_node*. As described in Section 3.3.1.1 for the simple Stripe Scan operator, at any given time, the current cursor can be interrupted, and a new cursor is opened, locating a node *after* the current node and thus skipping all intermediate nodes in the Stripe. Note that whenever a request for a new context node occurs, it may be the case that the new node region starts before the current node *cur_node*. In order to handle such cases, the Axis Stripe Scan operator in addition to Stripe Scan operator, may open a new cursor for locating any node that occurs *before cur_node*. In general, the Axis Stripe Scan operator may locate any node *n* of the Stripe having $n.start \geq hint\_val$, where *hint_val* is a value that is computed according the to axis semantics. To that end, it may (a) simply return the current node when $hint\_val = cur\_node.start$, (b) scan forward when $hint\_val > cur\_node.start$, or (c) locate a node that occurs before current node when $hint\_val < cur\_node.start$.

Axis Merge Scan This is the equivalent of merging Stripe Scan Operators, only that it operates in filter mode. Multiple Stripes are scanned as one and a node that satisfies the axis predicate for a context node is returned; otherwise, node ε is returned. It applies the same logic as the Axis Stripe Scan operator, only that since the input is retrieved from multiple Stripes, a merge operation is in need for producing nodes in document order with respect to the context node.

The retrieval of nodes contained in Stripes occurs in the same manner as in the single Stripe operator: Each time a new context node is used for a next(*ref*) call, the first node, in document order, matching the axis predicate is returned or node ε otherwise. Subsequent calls for the same context node will return any remaining nodes that satisfy the axis predicate. When a next() call occurs for a new context node, then the first node (in document order) will be returned with respect to the new context node, ignoring any other matching nodes for the previous context node.

3.3.3.3 Path Expressions in Filters

This is the operator that evaluates a path expression E_1/E_2 within a predicate expression. Since all location steps included in expressions E_1, E_2 are evaluated at Scan operators, we prefer the term of *Filter-Path* operator instead of the structural join term, stressing that its task is restricted to produce matching nodes for the path expression or ε node in the case where none node is produced. A Filter-Path operator, has two input operators; the left input operator provides the result of expression E_1 which in turn acts as context nodes (inner focus) for the evaluation expression E_2 while the right input operator provides the result of expression E_2. The process is outlined in Algorithm 3.7. Each invocation of next(*ref*) method produces a node that is part of the final result node sequence of expression $ref/E_1/E_2$; reference node *ref* acts as the context node in the outer focus for the evaluation of E_1. As soon as a node is produced for E_1 (left input operator), it serves as the context node in inner focus for the evaluation of expression E_2 (right input operator).

In detail, the process involves two distinct operations: (a) the context node (e_1) retrieval (lines 4 - 10) and (b) the output node (e_2) retrieval (lines 11 - 16). The context node retrieval is performed by the left input operator and it is always retrieved with respect to reference node *ref*, acting as the context node in outer focus for evaluating expression E_1 (line 6). In the case that no context node is retrieved, the process terminates and returns the empty node ε (lines 7 - 10), indicating that the evaluation of the path expression for context node *ref* does not produce any result nodes. If a context

Algorithm 3.7: path_flt::next (*ref*)

```
1  begin
2      if (ref_pos <> ref.start) then state ← NEXT-CTX;
3      repeat switch (state) do
4          case NEXT-CTX
5              ref_pos ← ref.start;
6              e1 ← left_input.next(ref);
7              if (e1 = ε) then
8                  e2 ← ε;
9                  exit ← true;
10                 break ;
11         case NEXT-CND
12             e2 ← right_input.next(e1);
13             if (e2 <> ε) then
14                 state ← NEXT-CND;
15                 exit ← true;
16             break ;
17     until (exit = true);
18     return e2;
19 end
```

node (e_1) is actually produced from the left input operator, it acts in turn as the context node in inner focus for the evaluation of expression E_2 (line 12). If a non-empty node (e_2) is produced from the right input operator, it means that there exists a match for the path expression and the output node is returned. Subsequent invocations of the next() method, will simply produce the rest of the matching nodes for the same context node e_1. As soon as they are exhausted, a new context node will be produced and the process will continue in a similar manner. Note, that at any time, a call of next() with a different reference node (outer focus context node) causes a new inner focus context node to be produced (line 2). This effectively means that a Filter-Path operator does not necessarily produce the complete result node sequence for the step expression, but it provides with a partial evaluation satisfying the predicate semantics.

3.3.3.4 Predicate Expressions in Filters

This operator evaluates a predicate expression $E_1[E_2]$ within a predicate expression. We term this as a *Filter-Predicate* operator and its task is restricted to produce nodes from the E_1 expression that satisfy predicate expression E_2.

The process is outlined in Algorithm 3.8. Each invocation of the next(*ref*) method produces a node that is part of the final result node sequence of expression $ref/E_1[E_2]$.

Algorithm 3.8: filter_flt::next (*ref*)

```
1  begin
2      repeat
3          e1 ← left_input.next(ref);
4          if (e1 = ε) then exit ← true;
5          else
6              e2 ← right_input.next(e1);
7              if (e2 <> ε) then exit ← true;
8      until (exit = true) ;
9      return e1;
10 end
```

Similar to the Filter-Path operator, reference node *ref* acts as the context node in the outer focus for the evaluation of E_1. As soon as a node is produced for E_1 (left input operator), it serves as the context node in inner focus for the evaluation of predicate expression E_2 (right input operator).

Again, the process involves two distinct operations. The predicate context node retrieval is performed by the left input operator and the context node is always retrieved with respect to reference node *ref* (line 3). In the case that no context node is produced, the process terminates, returning the empty node ε (line 4). If a context node (e_1) is actually produced from the left input operator, it provides the context node in inner focus for the evaluation of predicate expression E_2 (line 6). If a non-empty node (e_2) is produced from the right input operator, it means that there exists a match for the predicate expression and the context node is retained. On the other hand, the ε node verifies that the predicate expression is not satisfied for context node e_1 and it is thus discarded. Note, that unlike the Filter-Path operator, in the case of a produced node e_2 for the predicate expression, we do not produce any further results. We merely continue with the next context node e_1. This demonstrates that a Filter-Predicate operator does not produce the complete result node sequence for the predicate expression.

3.4 Stripe Aware Optimisation

In Section 3.3, we presented algorithms for the efficient evaluation of location steps. The main concern was that each location step should provide an ordered result without containing duplicate nodes. It is already shown [47, 49, 50] that tree-aware evaluation algorithms benefit in terms of I/O over algorithms that are unaware of the underlying tree-structure. Our evaluation algorithms were designed taking this property

into account. As described in Section 3.2, our striped model can provide with prior knowledge of properties, such as fwd and single_level properties, for an operator's input Stripes. We reason whether and under which circumstances, such information, extracted from our striped representation, can further provide optimisation opportunities. For the structural join operators, some of the algorithms can be replaced by others that are more more efficient and require less memory. In addition, when handling predicate expressions, we consider algorithms that fully facilitate our striped model for exactly retrieving the result nodes and thus reducing the overall I/O, whenever possible. We term such optimisations as *Stripe-aware* optimisations in analogy to the tree-aware optimisations, already proposed.

3.4.1 Structural Join Optimisations

A structural join operator has two input operators: The node sequence produced from the left input operator provides the context node sequence for the location step evaluation. The node sequence produced from the right input operator, provides a candidate node sequence according the location step expression semantics. Since the left input operator provides the point of reference for which the output nodes are selected, it is beneficial to consider whether prior knowledge of fwd or single_level properties for the context node sequence may lead to optimisation opportunities. We consider four location step axes: descendant, child, following and following-sibling.

3.4.1.1 Descendant Axis Structural join Optimisations

The naïve algorithm analysis for the descendant structural join operator, (see Section 3.3.2.1), explained that when the context node sequence contains nodes having an ancestor-descendant relationship, the evaluation of the descendant axis location step will result in a node sequence that is not in document order and that in addition contains duplicate nodes. Since, however, the produced node sequence for a context node is fully contained in the produced node sequence of its ancestor context node, the problem was resolved by considering only those nodes in the context node sequence that share a preceding-following relationship. Prior knowledge of the fwd or single_level properties regarding the context node sequence does not contribute to any further optimisation.

3.4.1.2 Child Axis Structural join Optimisations

The same does not hold for the child axis structural join operator though. Similar to the descendant axis counterpart, the naïve evaluation of the child axis location step for a context node sequence that contains nodes having an ancestor-descendant relationship, results in an unordered result node sequence. In order to tackle this problem, a range-partitioning technique was employed that maintains a stack of context nodes sharing an ancestor-descendant relationship for producing result nodes in document order.

On the other hand, it is shown that processing context nodes that share the preceding-following relationship, does produce result nodes in document order. Hence, prior knowledge of the fwd property for the context node sequence can further optimise the evaluation of the child location step. Indeed, when the fwd property holds for the context node sequence, it renders the range partitioning technique meaningless, only contributing to extra memory space required and computational cost. Instead, the descendant axis structural join algorithm can be used; each context node is processed, producing its descendant nodes. Note that the Input Minimisation process (described in Section 2.3) for the child axis location step will eliminate any input Stripes that provide any descendant nodes except the children nodes of the context nodes. This eliminates the need for sibling-based skipping when retrieving candidate nodes after a first child node match has been produced. Any node lying within the context node's candidate range is guaranteed to be its child node.

3.4.1.3 Following Axis Structural join Optimisations

For the evaluation of the following axis structural join, we merely need to identify the context node having the minimum *end* value, as described in Section 3.3.2.3. If the fwd property holds for the context node sequence, the requested context node is the first node of the context node sequence. Thus the retrieval of context nodes can be restricted to a single context node.

3.4.1.4 Following-Sibling Axis Structural join Optimisations

For the following-sibling axis structural join operator, even when the fwd property holds for the context node sequence, the range-partitioning technique is still needed for producing result nodes in document order. Consider for example two context nodes c_1, c_2 where node c_2 follows c_1. If, for instance, c_2 is a descendant of a sibling node of c_1 (case 2b), then the resulting node sequence may not be in document order, as described

Algorithm 3.9: opt_fol_sibl_sj::next (*hint*)

```
1  begin
2      repeat switch (state) do
3          case NEXT-CTX
4              hint.start ← max(hint.start, lst_cnd.end + 1);
5              ctx ← left_input.next(hint);
6              if (ctx.end < cnd.start) then
7                  state ← CHECK-CND;
8                  break ;
9          case NEXT-CND
10             cnd_hint.start ← ctx.end + 1;
11             cnd ← right_input.next(cnd_hint);
12         case CHECK-CND
13             if (ctx.par = cnd.par) then
14                 lst_cnd ← cnd;
15                 state ← NEXT-CND;
16                 exit ← true;
17             else
18                 state ← NEXT-CTX;
19             break ;
20     until (exit = true);
21     return cnd;
22 end
```

in Section 3.3.2.4.

When, however, the stricter property single_level holds, not only all context nodes have the same *level* value but due to the Input Minimisation process, the candidate node sequence is bound to contain nodes with the same *level* value as the context nodes. In such a case, the algorithm for the following-sibling location step evaluation can be reduced to an algorithm in the same spirit as the descendant axis structural join operator, apart from its axis-related parts. The algorithm of the next() method is described in Algorithm 3.9. No candidate range partition occurs and thus each context node is processed in isolation. For each context node, the following-sibling nodes are located and produced. We expect all following-sibling nodes n of a context node c to occur just after context node c and thus be produced sequentially (*i.e.,* no skipping is needed), satisfying predicate $n.start > c.end$ (lines 10-11). We use this observation to determine whether candidate nodes are exhausted for the active context node without knowledge of its parent node's full structural information (as *par_end* value for the original operator Algorithm 3.4) or using safety nets depending the *level* value of the retrieved candidate node (as at the alternative operator Algorithm 3.5). In this case, as

soon as a candidate node is retrieved that occurs after the context node but does not have the same parent node as the context node, we know there are no other following-sibling nodes for the active context node and thus we retrieve the next context node (line 18). Another interesting observation is the way the optimised algorithm prevents duplicate result node generation. Recall that duplicate nodes are produced from sibling nodes in the context node sequence. We avoid the duplicate node generation by skipping all context nodes that are siblings of any of the context nodes processed so far. In fact, we can avoid the extra cost of identifying the sibling relationship between context nodes by using the last produced result node for the active context node (line 4); after all, we produce sibling nodes of the active context node. This effectively skips all context nodes that would produce duplicates.

3.4.2 Filter Processing Optimisations

We have already presented a generic, context-node driven, iterative method for the evaluation of predicate expressions that (a) does not necessarily produce the complete result of a predicate expression (b) does not require extra sorting or duplicate-removal operations (c) supports any kind of node relationships, and (d) it is fully pipelined, producing minimal result for the predicate expressions.

We now present potential inefficiency issues when evaluating a location step as part of a predicate expression. As presented in Section 3.3.3.2, the evaluation of a location step in a predicate expression is handled by specialised, filter mode scan operators, the Axis Stripe Scan and Axis Merge Scan operators. Both scan operators, given a context node, process candidate nodes from their input Stripes and output those that match the axis relationship a with respect to the context node. In order to locate candidate nodes, each time a fraction of input Stripe(s) is scanned according to the context node's candidate range (as defined in Section 3.1.2). For instance, the candidate range for a context node u and the descendant axis is defined as $\mathsf{range}(u, descendant) = (u.start, u.end]$.

The problem that arises is that for certain axis relationships, namely the child and sibling relationships, the result node sequence when evaluating such relationships with respect to a context node is considerably smaller than the node sequence defined from the context node's candidate range. Consider for example context node c in Figure 3.3(b). Its candidate range for the child axis is defined as range: $(c.start, c.end]$, defining the candidate node sequence (d, e, f, g, h, i, j). However, the evaluation of the axis relationship for context node c, produces node sequence (d, f, j); a considerably

smaller result. Same observations can be made for the sibling relationships. For example consider a following-sibling axis location step for context node b in Figure 3.5(b). The result node sequence is (c,k) while the candidate node sequence resulting from candidate range: $(b.end, a.end]$, is (c,d,e,f,g,h,i,j,k,l).

This issue has been resolved for the location step evaluation algorithms by exploiting the sibling relationship among result nodes (for the child and sibling axis relationships). For instance, back to the child axis evaluation example for context node c, after producing result node d, node e can be skipped and node f is produced. Similarly, nodes g,h,i can be skipped and node j is then produced. An efficient implementation for the evaluation of such relationships in filter mode should also consider such optimisations.

However, because of the fact that the evaluation of predicate expressions occurs in a single context node, iteratively, this can result in many unnecessary initialisation/termination operations of cursors, either for skipping forward or for scanning overlapping candidate regions. Consider for example, the context node sequence (a,c) for the child axis location step. We begin by calling $\mathsf{next}(a)$ of the Axis Scan operator, which considers candidate range $(a.start, a.end]$ and opens a cursor for retrieving all nodes n, satisfying predicate $n.start > a.start$. Node b is produced. During the next invocation of $\mathsf{next}(a)$, node c is produced. The last child node of context node a, is located by skipping all descendant nodes of node c, *i.e.,* using a cursor for retrieving nodes n with $n.start > c.end$. When the evaluation begins for context node c, candidate region $(c.start, c.end]$ must be considered, which is included in the candidate region for context node a. Thus, when the first $\mathsf{next}(c)$ call is made, a new cursor must be opened to retrieve nodes n satisfying predicate $n.start > c.start$. Node d is produced while during the subsequent $\mathsf{next}(c)$ call, d's descendants are skipped and result node f is produced. Similarly, the subsequent $\mathsf{next}(c)$ call of the scan operator will skip f's descendant nodes and produce result node j.

Repetitive initilasation/termination operations of scan cursors may impact the evaluation time of the child or following-sibling axes location steps. However, providing Stripe-aware scan operators can in many cases resolve such issues. The key observation is that for both axes location steps, all nodes that belong to the result node sequence are sibling nodes. Sibling nodes, apart from the fact that they share the same parent node, they also share the same tree *level* value, which is easily derived from the active context node. This can further enhance the search criteria for locating children or following-sibling candidate nodes with respect to a context node.

In detail, when considering the child axis location step for a context node c, we need to locate those nodes n satisfying predicate $n.start \in (c.start, c.end] \wedge n.level = c.level + 1$. Similarly, for the following-sibling axis, the result nodes must satisfy predicate $n.start \in (c.end, c.par\_end] \wedge n.level = c.level$. Stripe Scan operators provide access to tree nodes though Stripes, ordered collections of XML tree nodes for each unique label path of the tree. This implies (Section 2.2) that all nodes of a Stripe S share the same tree *level* value, provided by function $\mathsf{level}(S)$. When requesting for nodes, thus, with a specific *level* value, it seems reasonable to restrict to input Stripes S that only contain nodes at the requested tree level, *i.e.,* $\mathsf{level}(S) = level$.

It is evident that the need for such optimisations is meaningful when multiple Stripes are involved that have different level() values. When the input involves a single Stripe or multiple Stripes of the same level() value, the level-based predicate part for parent-child or sibling relationships is ensured by the Input Minimisation process, that selects the minimum set of Stripes needed for the query evaluation (as described in Section 2.3). On the other hand, when multiple input Stripes of different level() value are involved, an optimised Axis Merge Scan operator can be used for the evaluation of child or following-sibling location steps in predicate expressions. This operator introduces a *level*-based Stripe filtering process where input Stripes are grouped according to their level() value, and all Stripes that do not contain a node matching with the active context node, are discarded. The scan process then only involves a qualified set of Stripes and the final merging process produces a node that satisfies the axis predicate and maintains (local) ordering with respect to the active context node, *i.e.,* for each context node, the result nodes are produced in document order.

Let us consider again the above example, context node sequence (a, c) for the evaluation of the child axis location step. The first call of $\mathsf{next}(a)$ for the Axis Merge Scan operator, will consider candidate range $(a.start, a.end]$ but only for input Stripes S having $\mathsf{level}(S) = a.level + 1 = 1$. Node b is retrieved and subsequent invocations of $\mathsf{next}(a)$ will produce nodes c and k, since only Stripes at *level* 1 are considered. Note that all children nodes for active context node a were retrieved without touching extra nodes or performing any skip operations. The process continues by calling $\mathsf{next}(c)$, the next context node. Similarly, the retrieval concerns input Stripes S having $\mathsf{level}(S) = c.level + 1 = 2$ for candidate range $(c.start, c.end]$. Node d is immediately retrieved and added to the result as it matches the axis predicate criteria. Subsequent calls of $\mathsf{next}(c)$ produce nodes f and i, no other node is retrieved other than the ones produced.

To conclude, the optimised level-based Axis Merge Scan operator efficiently supports the evaluation of child or following-sibling location steps in a predicate expression, by selecting a minimal set of input Stripes that may produce a matching node with respect to an active context node. The selection of Stripes is based on the Stripe level() and it results in avoiding nodes that will not contribute to the result. This can also be accomplished by using skipping techniques which although are in favour over naïve approaches that scan large Stripe fragments for locating a matching node, come at a cost, especially when extensively used. The optimised Axis Merge Scan operator minimises the use for skipping by providing direct access to the matching nodes. In addition, by selecting the appropriate set of Stripes it minimises re-scanning of Stripe fragments which in many cases reduces the overall I/O cost. at the expense of extra computation.

3.5 Related Work

There exist three main approaches for query processing in native XML systems: a) The *navigational* approach, b) the *binary join* approach and c) the *holistic twig join* approach.

3.5.1 Navigational Approach

Most navigational evaluation approaches are coupled with a graph-based structural summary that reflects the structure of an XML tree. (see Section 1.1.3.1). To that end, XML queries can be evaluated by navigating directly the original XML tree [38] or a compressed instance of the tree (*e.g.,* Dataguides [45], Skeleton [20, 19]). However, when large XML trees are considered, their structural summaries are either too large to fit in main memory (*e.g.,* Dataguides [45]) or inaccurate, requesting a validation post-processing step (*e.g.,* A(k)-index [61]). In addition, while navigation of structural summaries may favour the evaluation of the direct containment relationship, *i.e.,* parent-child relationship, it is rather expensive for evaluating ancestor-descendant relationships as it may result in scanning large parts of the graph summary, that are not relevant to the result.

To that end, the disk-based *F&B index* was proposed [102], that provides a disk-based structure of the F&B graph-based index proposed in [60]. The disk-based *F&B index* effectively lifts any main memory limitation and provides support for navigating

the XML structure. In addition, due to its tag-based node clustering in disk pages, it overcomes the navigation-inherent problem of accessing large parts of the original document tree, during the evaluation of ancestor-descendant relationships.

Many native XML DBMSs such as *Lore* [71, 72], *Natix* [39], and *Niagara* [77, 51], have been proposed that serialise the structure of an XML tree (or a collection of XML trees) in disk-based data structures that permit structural navigation. However, as already described, for the efficient evaluation of structural relationships, especially descendant-ancestor relationships, these systems do not rely on navigational evaluation but on structural join operations using B^+-tree node indexes.

3.5.2 Binary Join Approach

The binary join evaluation approach relies on the decomposition of an XML query to multiple pair-wise join operations to evaluate structural relationships. For this reason these join operations, are known as *structural joins*. Structural joins involve the encoding of XML nodes using specialised labelling schemes that capture the structural information of XML nodes. Node relationships are then processed by comparing node encodings. In general, a structural join algorithm accepts two input lists A and B of sorted nodes and structurally joins pairs of nodes (a,b) from both lists that satisfy the structural relationship. We now present the most important structural join proposals.

The *Multi Predicate MerGe JoiN*, MPMGJN [105], was the first approach of a structural join implementation handling node containment. While reasoning whether relational database systems are adequate for supporting efficiently such operations, the authors concluded that a main factor that doomed the RDBMS engine to suboptimal performance was the traditional merge join algorithm used for evaluating the containment property. The MPMGJN algorithm on the other hand, used by Information Retrieval (IR) engines, achieves significantly better performance due to multiple predicate merge criteria imposed, derived from the structural information of XML nodes. In detail, for each a node from list A, a range scan (inner-loop) $(a.start, a.end)$ occurs over list B, accessing only those b nodes contained in a. This results in fewer node retrieval and comparison operations among nodes of the input lists. Apart from the traditional merge join algorithm though, in the presence of B^+-tree indexes, a relational optimiser may choose to use an index nested-loop join operator, which exploits the index structure and performs the inner loop of the MPMGJN algorithm by probing the B^+-tree (performing a search operator and then scanning forward). It was shown that when input list

A is rather small in comparison to input list B, the index nested loop join outperforms the MPMGJN operator since by probing the index structure, it skips many unmatched nodes from list B. On the contrary though, when list A is comparable or larger than list B, the repetitive probing of the B^+-tree index will result in performing random I/O.

Similar to MPMGJN, the εε-Join [63] was proposed as an alternative to navigational approaches for processing regular path expressions on XML. To that end, any complex path expression can be decomposed into several simple path expressions, evaluated using εε-Joins, while their results are later combined (joined) using said join operators. An εε-Join operates on a nested-loop fashion (similar to the MPMGJN), performing a range scan of list B (inner loop) for each a node from list A (outer-loop). Another binary structural join algorithm proposed is the *ZigZag Join* [51]. The ZigZag Join was proposed when mixed node evaluation, *i.e.,* navigational with structural join approaches, was considered for XML query processing. The ZigZag Join maintains the core nested loops algorithm of the MPMGJN and further extends it by using an index structure for skipping over parts of an input list that is guaranteed that does not contain matching nodes on the other list.

The algorithms described so far, are all members of the *Tree-Merge* family of structural join algorithms as described in [8]. It is shown that while Tree-Merge structural join algorithms are usually superior to navigational approaches, they can not guarantee I/O optimality for two reasons:

(a) As already described, any Tree-Merge join algorithm during inner-loop processing, retrieves all b nodes from the B list that lie within range $(a.start, a.end)$. This, however is suboptimal when a parent-child relationship is involved, since in most cases node-set $\mathsf{children}(a)$ is a small subset of node-set $\mathsf{descendant}(a)$ and thus many nodes are retrieved although they do not contribute to the result.

(b) The nested-loop approach results in rescanning b nodes from list B when there is an ancestor-descendant relationship among nodes of list A. This holds when either the ancestor-descendant or parent-child relationships are evaluated.

To tackle such issues, the *Stack-Tree* family of structural join algorithms was proposed [8], motivated from the observation that a depth-first traversal of a tree can be performed in linear time when using a stack of nodes that extends up to the height of the tree. Such stack-based algorithms maintain a stack of a nodes (from list A) satisfying the ancestor-descendant relationship, *i.e.,* each node in the stack is a descendant of each node below it. This simplifies the identification of descendant nodes b from the

B list, since if a *b* node is found to be a descendant of the current top of the stack, it immediately implies that it is also a descendant of any other *a* node in the stack. This effectively guarantees I/O optimality since a single scan of each input list is performed. The suboptimality of the parent-child relationship is also resolved but only when the join operator involves the complete *A* list; if some *a* nodes are selected out of the list, then the algorithm suboptimality still holds.

Structural join algorithms produce (a,b) node pairs satisfying the containment relationship. Tree-Merge and Stack-Tree algorithms come in two flavours; when the result is ordered according to descendant nodes (*B* list), we have the Tree-Merge-Desc and Stack-Tree-Desc algorithms, as opposed to Tree-Merge-Anc and Stack-Tree-Anc algorithms that produce the result ordered on the ancestor nodes (*A* list). Experimental results in [8] demonstrate that both flavours of the Tree-Merge family have comparable performance which relies on the actual tree structure and the selectivity of the relationships tested. For the Stack-Tree family though, the Stack-Tree-Desc outperforms its Anc counterpart in all cases. This occurs because the Stack-Tree-Anc algorithm needs to materialise a fragment of result pairs due to result ordering. At any given time of the execution of Stack-Tree-Anc algorithm, the node at the stack bottom must be first produced before any other *a* node in the stack (having a *start* value smaller than any other *a* node in the stack). This means that for producing a node pair (a,b), we need to make sure that all descendant *a* nodes of the one on stack bottom along with all descendant *b* nodes are processed. These pending result pairs must be temporary materialised though, rendering the operator (partially) blocking.

As already described, for evaluating a path expression, this is usually decomposed into sub-expressions and in the end, a series of binary joins must be performed; the result of each binary join becomes input for its parent join operator. Join algorithms produce the join result ordered in either ancestor or descendant nodes and enable an XML query optimiser to enumerate alternative evaluation plans. In [103], the structural join order selection process is extensively studied, in a similar manner as a relational optimiser considers join order selection. The authors propose various query optimisation algorithms, such as dynamic programming-based algorithms that explore all plans available in the search place or other variants, enhanced by pruning techniques that remove evaluation plans leading to suboptimal solutions. Other proposed algorithms use heuristics that reduce the search space of available plans, for speeding up the optimisation process at the cost of a lower, but still acceptable, quality of the selected evaluation plan. Experiments conducted in the *Timber* XML database [55], demon-

strate that an optimisation algorithm that only considers non-blocking structural join operators, provides a good, although not always the best, evaluation plan. Such an optimisation algorithm, called "FP" for *fully pipelined*, is also time efficient compared to the other optimisation algorithm alternatives since the number of fully-pipelined produced plans is a small subset of all plans in the search space. Another interesting result is that when dealing with large datasets, the optimisation algorithms that explore the whole search space, select a fully-pipelined evaluation plan (as the FP algorithm). This happens because when dealing with large datasets, we expect the intermediate results of some join operation to be large as well. This means that blocking plans, for example plans containing sort operations, will result in sub-optimal solutions since they will need to materialise large intermediate results. Therefore, the best plan is always a fully-pipelined evaluation plan.

An extension of the Stack-Tree-Desc algorithm in the presence of indices on the input lists is described in [28]. The Stack-Tree-Desc Structural join algorithm processes sequentially both ancestor and descendant lists. However, if either input list is very selective, skipping opportunities arise. The *AncDesBtree* algorithm assumes that both ancestor and descendant node lists are B^+-trees indexed on *start* value (or that there is a single B^+-tree on $tag, start$ that provides input for the input lists). When a skip opportunity arises, it utilises the index structure to skip unmatched nodes. In that sense, we regard the B^+-tree-based AncDesBtree algorithm as an extension of the Stack-based structural join algorithm of [8] in analogy to the ZigZag Join [51] being the extension of MPMGJN [105] with skipping operations. Both algorithms extend their original join algorithms by skipping parts of input lists that do not contain matching nodes in the other list. Skipping is performed by probing B^+-tree indexes.

The structural join algorithms presented so far, focus on containment relationships, *i.e.,* ancestor-descendant or parent-child relationships. In an attempt for enhancing the relational database engine for efficient XPath query evaluation, the *Staircase Join* was proposed [49] as a "tree-aware" structural join algorithm that exploits the tree-structured XML data model and node relationships, to efficiently evaluate XPath location step expressions. The Staircase Join was developed on the foundation of an index proposal [47], designed to support XPath queries, thus more than regular path expressions which barely involve containment relationships among XML nodes. This work introduces notions like XML *document partitions*; any context node c defines four partitions according to the four major XML axes: descendant, ancestor, following and preceding. These partitions are disjoint and their union contains all nodes

of the XML tree where context node c belong to (except itself). Document partitions provide useful insight regarding candidate nodes that are produced when a location step expression is evaluated. This led to the definition of *query windows*; each XPath axis corresponds to a query window which is effectively translated in a multi-predicate region query for selecting nodes that satisfy the XPath axis. The Staircase Join generalised these concepts and provided a family of structural join algorithms that given a context node sequence and an axis, produce the result sequence of nodes according to axis navigational semantics. The Staircase join uses techniques like pruning, partitioning and skipping to efficiently evaluate location step expressions accessing at most $|context| + |result|$ nodes [47]. These techniques are successfully used on top of traditional RDBMSs [47, 49, 50] as well as main memory DBMS, (*MonetDB* [16]) [50].

Staircase Join was the first structural join algorithm that considered all XPath axes. Other proposals that also considered horizontal node navigation and in particular the sibling node relationship, are [97, 94]. Both pieces of work are based on the concept of *context sibling lists* for avoiding multiple passes over the input lists. According to this, all context node siblings are conceptually associated in a linked-list. This way, when a candidate node is retrieved, it is sufficient to check the sibling property against a single context node in a list. If these nodes are siblings, then the candidate node is a sibling node of any of the nodes in the context sibling list. In [94], a sibling list is retained for each unique $level$ value, while in [97], which is a variation of the stack-based structural join algorithms, sibling lists are linked to the context nodes being buffered on the stack.

We conclude this section with hybrid evaluation approaches that combine both navigational traversal and binary joins, verifying the pros and cons of each evaluation paradigm. In [51], in addition to a disk-based index that enables tree navigation, a structural join algorithm (similar to MPMGJN) was also employed; this operates on tag-clustered inverted lists to evaluate both parent-child and ancestor-descendant relationships. Both theoretical and experimental results demonstrated that the navigational approach is preferred for evaluating parent-child relationships while for ancestor-descendant relationships, the join approach is superior. A similar study was later conducted in [106] in which the *next-of-kin (NoK)* pattern tree was proposed, where the navigation style evaluation is explicitly preferred for handling parent-child relationships, whereas the join approach is selected for ancestor-descendant relationships.

3.5.3 Holistic Twig Join Approach

The third evaluation approach that is extensively used is that of holistic twig joins. Twig pattern queries is a class of XML queries that uses a tree-structured pattern to capture structural relationships of the requested data. Binary join algorithms provide support for the evaluation of twig patterns but they may produce large intermediate results that do not contribute to the final result of the twig pattern. To address such problems, a holistic twig join algorithm was proposed, *TwigStack* [18], to minimise useless intermediate results. The TwigStack algorithm uses a chain of linked stacks, one for each node in the twig pattern query, which provide a compact representation of root-to-leaf twig partial results. These are then merged to compose the final twig query result. The main feature of the TwigStack algorithm that prevents large intermediate results being produced is that a node is pushed onto the stack S_q of a twig node q (and thus is considered to produce results) only if there exist XML nodes in each node of its sub-twig pattern, that satisfy the ancestor-descendant relationship. This condition does not fully eliminate redundant results, however it reduces the large intermediate results produced by binary join approaches. To that end, TwigStack is optimal in terms of intermediate results, only when twig pattern queries contain ancestor-descendant node relationships. When parent-child relationships are also present, *i.e.,* for twig patterns with mixed parent-child and ancestor-descendant relationships, no holistic twig join algorithm employing sequential scan over the input lists can ensure optimality [30].

Many subsequent pieces of work optimise TwigStack in various ways. TSGeneric [56] is a TwigStack variation that uses statistical information to skip unmatched nodes and thus reduce its I/O cost. Both approaches were also enhanced by specialised B^+-tree indexes (XB-Trees [18], XR-Trees [57]), that index node ranges and provide skipping capabilities on input lists. TwigStackList [65] also optimises TwigStack when parent-child relationships are involved. Twig^{2}Stack [23] and TwigList [84], reduce the cost of the TwigStack merging phase. However, the memory requirements for these approaches can be very high. Other holistic approaches depart from region-based schemes and rather use prefix-based, Dewey Decimal schemes which allow them to reduce the number of the input lists for evaluating a twig pattern [66]. Finally, in [25], various partitioning schemes are considered for the holistic evaluation of twig pattern queries.

3.6 Discussion

As already presented in this chapter, we preferred the binary join approach for evaluating $\mathcal{XP}$ queries, over the other two alternatives. Our striped storage model provides the infrastructure for selecting the appropriate input lists for binary join operations. These lists contain XML nodes that are relevant to the specified node relationship. Each input list is identified during the *Input Minimisation* process, described in Section 2.3. In addition, for the evaluation of subsequent parent-child relationships, where the navigational approaches are superior to the binary joins, our striped representation provides query rewriting opportunities; subsequent parent-child operations are substituted by a single ancestor-descendant join operation, operating on relevant node lists. This results in minimising query input in addition to the number of binary join operations.

The holistic twig join approach is considered the state the art in evaluating XML queries. The twig query is evaluated by a single operator, the twig join operator. Its optimality compared to the binary approach, is ensured from the fact that the algorithm is based on the containment property among XML nodes in each of the tree-structured twig pattern. As such, it can efficiently process ancestor-descendant and parent-child node relationships but it remains unclear whether such an approach is feasible when arbitrary node relationships are involved. The only relevant piece of work in this direction is that of [67], where the problem of the holistic evaluation of ordered-based node relationships is also considered. For an ordered twig pattern, apart from the containment condition between the connected query pattern nodes, the order condition is also considered between sibling query pattern nodes and thus order-based node relationships can also be evaluated, addressing a larger fragment of XPath expressions. Despite that, not any node relationship combination can be evaluated as a single twig join operator, due to the restrictions imposed among the query pattern nodes. On the other hand, the binary join approach provides generic, decoupled operators that may evaluate any node relationship and pipeline their results to other binary join operators that process node relationships independently.

Our binary join operators are implemented taking into consideration features from several approaches that incrementally addressed many issues. In detail, we have used expertise of already proposed, state-of-the-art algorithms such as the Stack-based structural join algorithms [8] and the tree-aware staircase-join algorithms [47, 49]; these are further enhanced, whenever possible, by exploiting our striped storage model.

We refer to our XML query engine as being “lightweight”, a term borrowed from [76]

to describe query engines that do not incorporate cost-based optimisation modules. In lack of such optimisation modules in addition to statistical information, which we consider for future work, we strive for evaluation plans that do not employ blocking operators and thus do not need to materialise intermediate results. Our evaluation algorithms operate in ways that produce duplicate-free output, sorted in document order, and thus eliminate the need for sorting or/and duplicate removal operations. To provide such evaluation plans, we decided to support predicate expressions by a context-node driven, iterative approach that a) does not produce the full structural join result and b) produces sorted output. Our intuition in using fully pipelined evaluation plans is verified by the work in [103], where the problem of structural join order selection is considered, as described in Section 3.5.2. Given the conditions described above and since no optimisation module is used, the generation of evaluation plans is a direct mapping of conceptual query operators to physical query operators. As for the conceptual query plan, it is deduced directly from the XPath abstract syntax tree in polynomial time (the size of the tree). If n is the number of step axis expressions of an XPath expression p, then the optimisation complexity is at most 2^{n-1} since for most structural join operators, there exist two implementations (the Stripe-aware and Stripe-unaware implementations). However, we do not perform any kind of exploration, we simply traverse the conceptual query plan and choose the optimised join algorithm. In practice though, the number of structural join operators and thus the number of decision points is usually much smaller than $n-1$ due to Path Minimisation.

A direction of future work is to implement the reverse axis structural join operators, in addition to their forward axis counterparts. Nevertheless, we are still able to process $\mathcal{XP}$ queries that contain reverse axis steps, by applying rewriting rules, as defined in [81]. These rules transform an XPath expression that contains reverse axis location steps into an equivalent expression that contains a minimum set of forward axis steps. However, it is interesting to implement the reverse axis structural join operators, as it not clear if the forward-only equivalent expressions can be evaluated more efficiently than the original expressions that contain reverse axis location steps. Finally, for the path expression evaluation, we did not consider positional predicates. For supporting those, we can enhance our structural join algorithms with the techniques described in [98]. For this purpose, we need to disable any kind of skipping on the context node sequences, since the position of a candidate node with respect to a context node is now significant.

Another interesting direction for future work is to incorporate holistic twig join op-

erators in our query engine. Our striped model can effectively provide the relevant lists of XML nodes involved in twig join evaluation. Identifying relevant input lists has also been discussed in [25, 11]. In [25], various partitioning schemes of XML documents, called *Streaming Schemes*, were considered for the evaluation of twig pattern queries. A pruning process, in the same spirit as our *Stripe Pruning* (Section 2.3), is applied so that certain streams that are not relevant to the query are pruned, before evaluating the twig pattern. Experimental results demonstrate that certain refinements of *Tag Streaming* such as the *Prefix Path Streaming, PPS*, which is essentially our proposed decomposition, can further reduce irrelevant parts of the XML documents considered for the evaluation of twig patterns. In addition, our *Path Minimisation* process (Section 2.4) can be applied to reduce the size of the twig pattern query, by removing internal nodes of the twig pattern. This, apart from reducing the query input size, it can reduce the amount of intermediate results, produced by the twig join algorithm.

Chapter 4

Explicit Storage Scheme

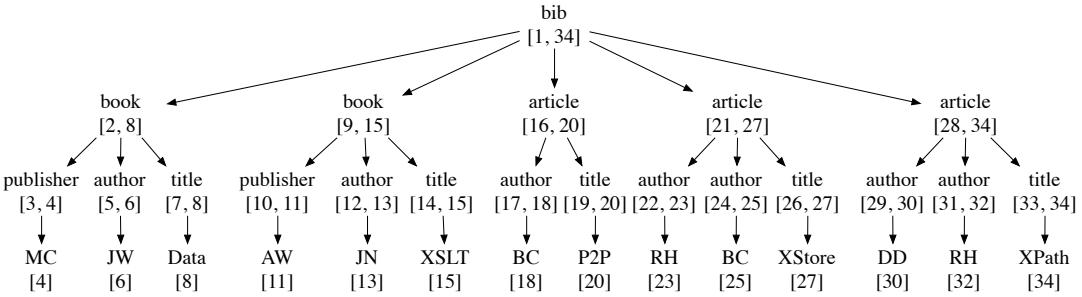
Having a general storage decomposition technique and a query evaluation pipeline in place, we now proceed to the definition of specific storage schemes for Striping XML.

In this chapter, we present the first and most natural application of the striped decomposition model. The rest of the chapter is organised as follows: In Sections 4.1 and 4.2 we present the most natural storage scheme for striped XML and present its storage characteristics. In Section 4.3, we describe the loading process of an XML document to our native XML store, while in Section 4.4, we describe how node sequences can be derived from Stripes. Finally, in Section 4.5, we present an extensive experimental study, stressing various aspects of the proposed XML store and a comparison with the state-of-the-art in XML query processing system.

4.1 Shredding XML

In this section, we present the first and most natural application of the striped decomposition model. We apply our generic decomposition process as described in Section 2.2: We create a single Stripe S_p for each unique label-path p and XML nodes are shredded according to their label-path value p. To that end, each XML tree node $n \in p$ (*i.e.,* $\mathsf{path}(n) = p$) is explicitly stored as a single Stripe node $N \in S_p$. There exists a 1-1 relationship between a tree node n and a Stripe node N, meaning that each tree node n that satisfies path p corresponds to a unique Stripe node N of Stripe S_p and vise-versa.

As described for the general Striping model, *Path Stripes* are structures that allow us to index directly into the structure of the XML document. A Path Stripe of a label-path p is defined as a set of Stripe nodes that correspond to the set of tree elements n satisfying label-path p. However, to efficiently support query evaluation



(a) An example XML document tree

/bib/book

[s,e]	par
[2,8]	1
[9,15]	1

/bib/article

[s,e]	par
[16,20]	1
[21,27]	1
[28,34]	1

/bib/book/author

[s,e]	par
[5,6]	2
[12,13]	9

/bib/article/title

[s,e]	par
[19,20]	16
[26,27]	21
[33,34]	28

/bib/book/title

[s,e]	par
[7,8]	2
[14,15]	9

/bib/article/author

[s,e]	par
[17,18]	16
[22,23]	21
[24,25]	21
[29,30]	28
[31,32]	28

(b) Path Stripes for the document tree in Figure 4.1(a)

/bib/book/publisher

start	text
4	MC
11	AW

/bib/book/author

start	text
6	JW
13	JN

/bib/book/title

start	text
8	Data
15	XSLT

/bib/article/author

start	text
18	BC
23	RH
25	BC
30	DD
32	RH

/bib/article/title

start	text
20	P2P
27	XStore
34	XPath

(c) Value Stripes for the document tree in Figure 4.1(a)

Figure 4.1: Explicit storage scheme for the striped model

and in particular structural joins, we regard Path Stripes as element node sequences satisfying document order. Each Stripe node is assigned the Region encoding triplet: $\langle start, end, par \rangle$ derived from the tree element node n it corresponds to. Therefore, to process structural predicates between XML elements, we apply exactly the same tests, as defined in Section 3.1.2, between their corresponding Stripe nodes. Other helpful attributes for the evaluation of structural predicates are the *level* and *label* values of the element nodes. Note that these attributes are not explicitly stored with the Stripe nodes as they can be derived from the Stripe path itself. We describe this later, in Section 4.4.

Apart from the pure structural part of the document which is covered by the set of Path Stripes $\mathcal{P}$, we extract the textual part of the document tree and store it separately. As in the general Striping model, we create an *Attribute* or *Value Stripe*, for each unique label-path p of T leading to an attribute or text value respectively. Each node of an Attribute/Value Stripe is stored as a $\langle start, text \rangle$ tuple, where the *start* value serves as its positioning in the document and *text* is the actual text or attribute value. Similarly to the Path Stripes, *level* and *label* attributes are not explicitly stored in Stripe nodes

but are derived for the Stripe path.

Example 4.1: For the document of Figure 4.1(a), a subset of Path and Value Stripes are shown in Figure 4.1(b) and Figure 4.1(c) respectively. Each node $n \in T$ now corresponds to a single Stripe node N in the appropriate Path or Value Stripe. □

4.2 Stripe Storage

All (Path, Attribute or Value) Stripes are stored in separate B$^+$-tree structures with the *start* value acting as the B$^+$-tree key. Stripe nodes are thus stored in ascending *start* value order which implies that they are stored exactly in the order their corresponding tree nodes n appear in the document *i.e., document order*. Thus, any Path, Attribute or Value Stripe is stored in document order.

In addition, we maintain metadata information for Stripes. This is kept in three separate B$^+$-trees, one for each type of Stripe; these B$^+$-trees comprise the *system catalog* and are used to map Stripe paths, *i.e.,* their path p, to internal B$^+$-tree system identifiers. Either B$^+$-tree can be searched for exact or approximate matches to Stripe paths, which is required for locating Stripes for XPath step expressions, as described in Section 2.3.2.

4.3 Document Loading

The XML document loading process is fairly straightforward for the explicit storage scheme. First an XML parser is needed for parsing the input XML document. XML parsers typically support two main categories of APIs; tree-based APIs such as DOM [80] and event-based APIs such as SAX [4]. The former category builds an in-memory tree representation and provides navigational access methods. The latter is used in a streaming fashion; as the input is consumed, parsing events (such as the beginning or end of elements) are being generated and callback procedures are invoked handling each event accordingly. The primary advantage of the event-based APIs is that they do not require the entire XML document to be stored in main memory; only the information about the node currently being processed is needed. This makes it possible to process large XML documents, without incurring a large memory cost. For the loading process, we merely require a single pass over the input document. Therefore, we chose the latter category of XML APIs and in particular SAX, as it operates

in a streaming fashion, requiring a small amount of memory regardless of the input document size.

The loading process performs two main tasks: The first is to locate the appropriate Stripe where each input XML tree node is to be stored. The second task is to assign the appropriate structural and/or textual information of the input tree nodes to Stripe nodes in order to serialise them in Stripes. For efficiently performing both tasks, we build a main memory tree, the skeleton tree or simply the *skeleton*, reflecting all unique label-paths of the document tree. The skeleton can be seen as a label trie, where each skeleton node is a label and any root to node label-path corresponds to a unique label-path of the original XML tree. Depending on the tree node type, skeleton nodes are labelled accordingly with: (a) the element tag name, for element nodes, (b) the attribute name prefixed with the '@' character, for attribute nodes, (c) the special label "#text", for text nodes, and (d) the special label "#root", for the document root node. The skeleton is built on the fly as tree nodes are parsed and new label-paths appear. When an element/attribute/text node is encountered, we first try to locate the appropriate skeleton node child by comparing its label value. If such a child exists, we know that we have already encountered this label-path and that the appropriate Stripes have been already created. On the contrary, if the skeleton child node does not exist, we add it, create the appropriate Stripe according to the node's label-path as well as its tree node type and navigate to the just added skeleton node. This way, at any given time, skeleton nodes have references to the Stripe locations (identifiers) and these are used when a Stripe node is ready for insertion. Attribute and Value Stripe nodes are being inserted upon retrieval by the XML parser. For Path Stripe nodes, when an element tree node is encountered (start_element SAX event), the current *start* value is kept at the skeleton node. As soon as the complementary event (end_element) is triggered, we can access its *start* value, the skeleton node's parent *start* value (for *par* value) and calculate the *end* value accordingly. We then insert the Path Stripe node to the Path Stripe appointed by the current skeleton node.

4.4 Node Reconstruction

We have already described that we use the Stripe abstraction to emulate node sequences during query evaluation (Section 3.2.1) and that tree nodes are encoded as tuples of the form: $\langle start, end, par, level, kind, label, text \rangle$. Stripe Scans are the operators responsible for accessing Stripes and, as a result, they are delegated to re-construct XML tree

nodes, encoded as described above, from Stripe nodes.

For the explicit storage scheme, this task is fairly straightforward: All Stripe nodes stored in the same Stripe S_p, share the same *level*, *label* and *kind* values; these are derived for the Stripe itself: $level = \mathsf{level}(S_p)$, $label = \mathsf{label}(S_p)$, while the *kind* value is the type of the Stripe (Path, Attribute or Value). For Path Stripes, the *text* value is null since they encode element nodes, while the *start*, *end* and *par* values are copied each time from the retrieved Path node N. On the other hand, for Attribute/Value Stripes, the *start* and *end* values are set to the *start* value of the retrieved Stripe node, while the *par* value is set to 0[1]. Finally, the *text* value is copied from the Stripe Node. To summarise, we have that a tree node n from a Stripe node N of Stripe S_p is produced as defined by function r:

$$\mathsf{r}(S_p, N) = \begin{cases} \langle N.start, N.end, N.par, \mathsf{level}(S_p), \mathsf{PATH}, \mathsf{label}(S_p), \mathsf{null} \rangle & \text{if } S_p \text{ is a Path Stripe,} \\ \langle N.pos, N.pos, 0, \mathsf{level}(S_p), \mathsf{ATTR}, \mathsf{label}(S_p), N.text \rangle & \text{if } S_p \text{ is an Attr. Stripe,} \\ \langle N.pos, N.pos, 0, \mathsf{level}(S_p), \mathsf{VALUE}, \mathsf{label}(S_p), N.text \rangle & \text{if } S_p \text{ is a Value Stripe} \end{cases}$$

4.5 Experimental Results

We now describe the experimental setup used to verify the efficiency of our native store prototype. Our prototype was implemented in C++. We used Berkeley DB [2] as the storage infrastructure for our native store and in particular its B$^+$-tree implementation for Stripe and Catalog storage. The C++ code was compiled with the GNU gcc compiler (version 4.3.3) using the -O3 compilation flag.

The hardware platform used for our experiments was a Dell Precision T5400 workstation, with an Intel Xeon E5420 quad-core processor, clocked at 2.5GHz, and 4GB of physical memory. The operating system was Debian 4 (64-bit version, kernel 2.6.26) and the filesystem hosting our repositories was ReiserFS.

We conducted our experiments over three well known datasets: Xmark, Mbench and DBLP. Their details are summarised in Table 4.1.

The Xmark XML dataset was proposed as part of the *Xmark* XML Benchmark project [88], to model an auction website. Xmark is a deeply-nested dataset, which mixes both regular structured elements with highly irregular data-oriented document parts that contain a great amount of mixed content and long textual values. For our ex-

[1]For evaluating element-attribute or element-value parent-child relationships, the *level* value is used along with their region encoding $(start, end)$.

Datasets	Xmark				Mbench			DBLP		
SF	0.1	1	10	100	0.1	1	10	1	5	10
Size	12M	112M	1.1G	11G	46M	496M	4.8G	99M	493M	985M
Paths	1046				178			276		
Tags	78				11			43		
Height	13				18			8		
Elements	0.17	1.67	16.7	167.1	0.07	0.74	7.29	2.61	13.05	26.09
Attributes	0.04	0.38	3.83	38.32	0.47	5.1	50.37	0.33	1.65	3.31
Text	0.3	3.02	30.31	303.23	0.14	1.48	14.58	5.22	26.09	52.17
WS Text	0.19	1.85	18.55	185.56	0.07	0.74	7.29	2.86	14.32	28.63
WS %	61.2				50			54.9		

Table 4.1: XML dataset statistics. All XML nodes are counted in millions

periments, we created documents using the `xmlgen` data generator and scaling factors 0.1, 1, 10 and 100, producing auction documents ranging from 11MB to 11GB.

The Mbench XML dataset was proposed as part of the *Michigan* XML Benchmark project, as an attempt to capture the rich variety and distribution of XML data structures. An Mbench XML dataset is deeply-nested, its structural part is relatively regular but is characterised by the high degree of its recursive structure. It merely uses a small number of element tags, two in particular, and exploits attribute values as a means to control structural and/or value selectivity. For our experiments, we created documents using the `mbgen` data generator and scaling factors 0.1, 1 and 10 producing documents ranging from 50MB to 5GB.

Finally, the DBLP XML dataset is an XML snapshot of a bibliography database of Computer Science journal and conference proceedings. Its structure is not as deep as the previous two datasets already described, while it is fairly regular. For our experiments, we created documents of scaling factors 1, 5 and 10 by copying the original XML snapshot as many times as the scaling factor implies and adding the resulting bibliography subtrees under a new top-level element `<db>`. The produced document's size ranges from 100MB to 1GB.

Our experimental study is divided in three parts, one for each of the Xmark, Mbench and DBLP datasets. We shredded all XML documents in our native store prototype; the size of the striped documents are presented in Table 4.2 while the resulting Stripe type distribution is depicted in Figure 4.2. Note, that the total number of all Stripes for each

SF	XML SIZE	SRX SIZE	PATH	ATTR	DATA
Xmark					
0.1	12M	30M	8.0M	1.4M	18M
1	112M	202M	44M	11M	145M
10	1.1G	1.9G	396M	107M	1.4G
100	11G	19G	3.9G	1.1G	14G
Mbench					
0.1	46M	59M	1.9M	12M	46M
1	496M	622M	18M	116M	489M
10	4.8G	6.1G	171M	1.2G	4.8G
DBLP					
1	99M	206M	63M	12M	132M
5	493M	1016M	307M	57M	653M
10	985M	2G	612M	114M	1.3G

Table 4.2: Striped storage for XML datasets

dataset is the number of unique document paths, as imposed by the Striping process. The experimental study for each of the datasets is organised as follows:

Impact of Striping We first aim to verify the effectiveness of our proposed striped model by demonstrating the impact of Striping on query input. One of the merits of Striping is that it provides the means to effectively minimise input so that the query engine touches only data that are relevant to the given query. This process, termed *Stripe Projection*, is part of *Input Minimisation*, introduced in Section 2.3.2. We first consider the actual *number of Stripes* being selected for query evaluation with respect to the total number of Stripes which comprise the striped

Dataset	PATH	ATTR	DATA	TOTAL
Xmark	515	34	497	1046
Mbench	28	123	27	178
DBLP	127	23	126	276

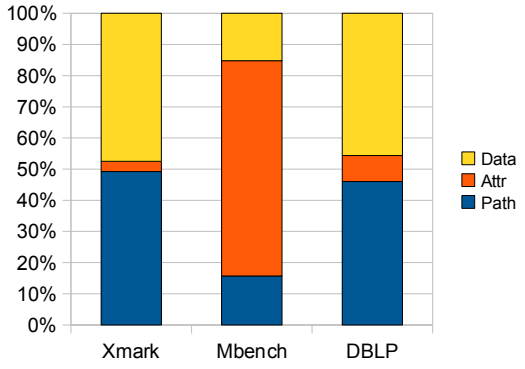


Figure 4.2: Stripe type distribution per dataset

document. However, the number of selected Stripes cannot always provide an absolute metric for query input, since the cardinality of a Stripe and thus its size is closely coupled with the distribution of document nodes at the document paths and/or the actual data stored in Stripes. The former is greatly influenced by the "shape" of the document tree, while the latter also concerns the type of nodes that are stored in a Stripe (*e.g.,* element vs text) and their size. To that end, in addition to the number of selected Stripes, we also measure the *Stripe cardinality i.e.,*, the number of nodes stored in the selected Stripes, as well as the *Stripe size*, *i.e.,*, the number of I/O pages required for Stripe storage. The cardinality and size of the selected Stripes are then compared to the total cardinality and size of all Stripes that comprise the striped document.

Impact of Stripe Pruning We then turn our attention to the impact of the *Stripe Pruning* and *Path Minimisation* processes on query input. Stripe Pruning, or simply Pruning (Section 2.3.3), is the complementary part of Stripe Projection; they both constitute the *Input Minimisation* process. Pruning operates on the selected (set of) Stripes as produced by the Stripe Projection process and further prunes Stripes (whenever possible) that are guaranteed not to contribute to the query result. Path Minimisation or simply Rewriting (Section 2.4), on the other hand, operates on path expressions and reduces sub-expressions by applying path equivalence rules that hold over our striped representation.

Both Pruning and Rewriting, may impact the number of selected Stripes for query evaluation. It is often the case that their result overlaps, *i.e.,* the effect of applying one process is, (partially) covered by the effect of the other. However, in other cases, their combined result has greater impact than the result they achieve when applied in isolation. For our experiments, we use the following notation: the selected StripeSet produced simply by applying Stripe Projection is $\mathcal{SS}_{NV}$ (naïve); the StripeSet produced by also applying Pruning or Rewriting is $\mathcal{SS}_{PR}$ or $\mathcal{SS}_{RW}$ respectively. Finally, the selected StripeSet produced by applying both Pruning and Rewriting (on top of Stripe Projection) is $\mathcal{SS}_{PR+RW}$. By definition, we have that $\mathcal{SS}_{PR+RW} \subseteq \mathcal{SS}_{PR}$, $\mathcal{SS}_{PR+RW} \subseteq \mathcal{SS}_{RW}$ and $\mathcal{SS}_{PR} \subseteq \mathcal{SS}_{NV}$, $\mathcal{SS}_{RW} \subseteq \mathcal{SS}_{NV}$. Similarly to the Striping impact results, apart from considering the actual number of Stripes being reduced by Pruning or Rewriting, we also consider the reduction of Stripe size, measured in I/O pages.

Impact of Operations Pruning Whenever Path Minimisation is applicable, apart from

reducing query input in terms of Stripes, we further benefit from the fact that certain operators are removed from the evaluation plan. In particular, there are two types of operations that are removed when Rewriting rules apply. The first is that of Scan operators, a side-effect of reducing the number of Stripes for query evaluation. Moreover, Structural Join operators are also removed since Scan operators for certain path expressions have been removed and thus there is no need for stitching up results. This effectively reduces the size of our evaluation plans.

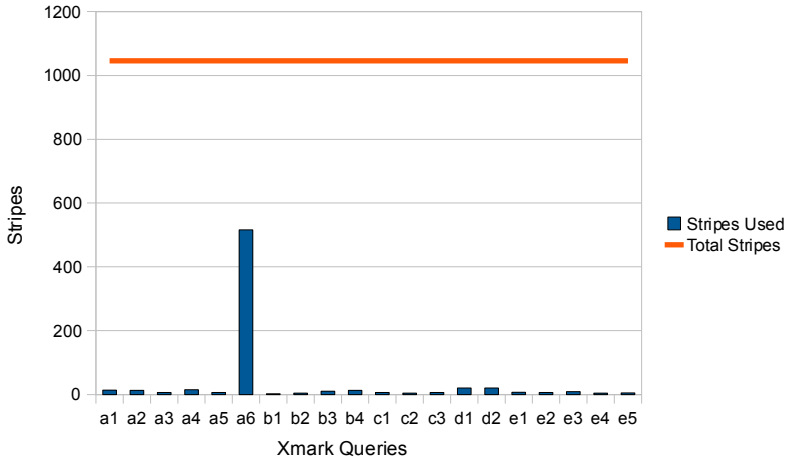
Query Performance We finally present the query evaluation performance of our native store prototype SRX. We compare the evaluation performance of applying none, some, and all of our optimisations, namely Pruning (PR), Rewriting (RW) and (Stripe-aware) Optimisation (OP). As already described, Pruning involves applying Stripe Pruning, Rewriting involves applying Path Minimisation, while Optimisation involves the use (whenever possible) of Stripe-aware evaluation algorithms as these are presented in Section 3.4. All possible combinations of Pruning, Rewriting and Optimisation renders eight possible evaluation setups. The two extreme cases are those that none and all optimisations are applied, the NV and PRO evaluation setups. Furthermore, for comparing the impact of the proposed optimisations, we apply them in isolation, rendering the PR, RW and OP evaluation setups. Finally, we consider the remaining combinations of optimisations, rendering PR+OP, RW+PR and RW+OP evaluation setups.

4.5.1 Xmark

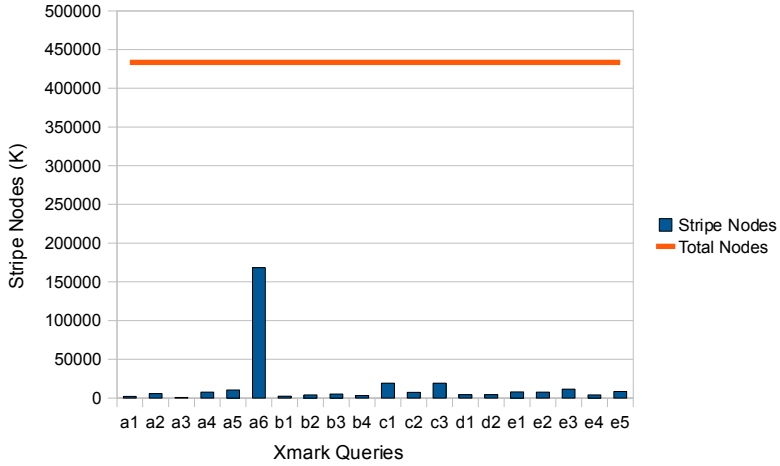
We now present the experimental results for the Xmark datasets.

4.5.1.1 Impact of Striping

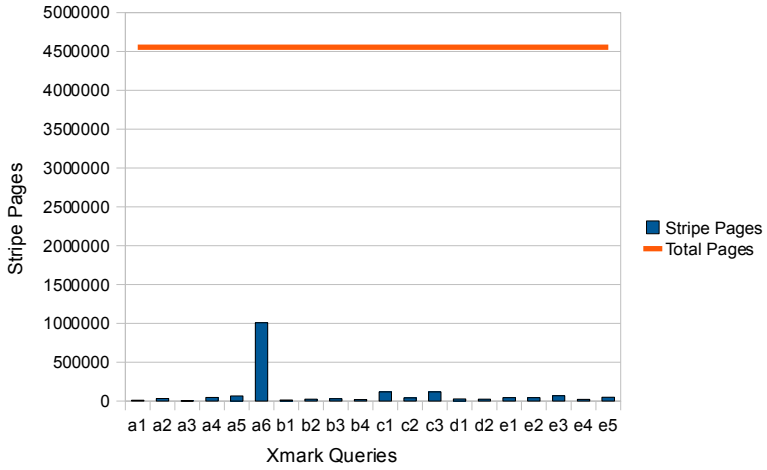
In this section, we aim to verify the effectiveness of our proposed striped model. Figure 4.3 depicts the impact of the Stripe Projection process, which we refer to as Striping, with respect to our testbed queries for the Xmark dataset. As seen from Figure 4.3(a), for the evaluation of each of the queries (with the exception of query a6), the number of Stripes that need to be accessed hardly reaches the 2% of the total number of Stripes of the striped dataset. Query a6 is an exception to that since it employs a "//"' operator which is the abbreviated syntax for expression "/descendant-or-self::node()/child", which selects all nodes of the document, and thus, the impact of



(a) Usage of Stripes



(b) Cardinality of the selected Stripes



(c) Size of the selected Stripes

Figure 4.3: Impact of Striping on the Xmark dataset

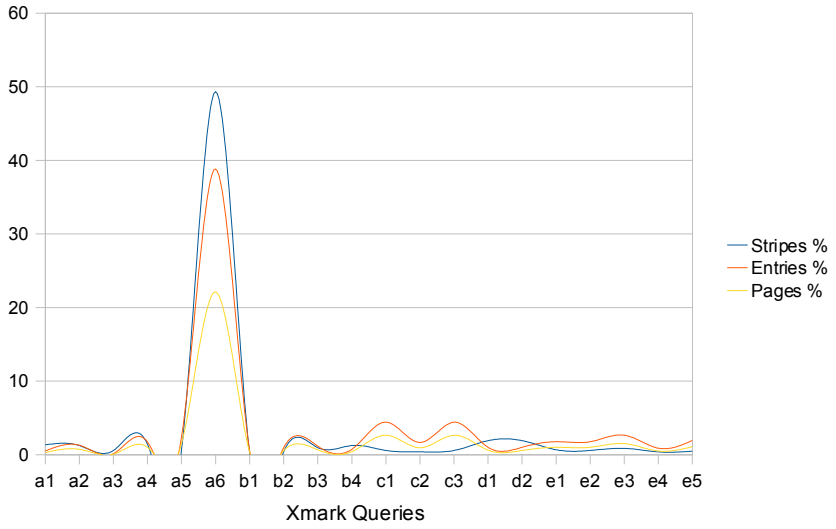


Figure 4.4: Normalised impact of Striping on the Xmark dataset

Striping would be minimal, in fact zero. However, we optimised this part by restricting access only to the Path Stripes and thus resulted in reducing input size by a factor of 50%.

The impact of Striping regarding the Stripe cardinality and size is depicted in Figures 4.3(b), and 4.3(c), respectively. As far as the former is concerned, the cardinality of the selected Stripes that are required for query evaluation follows the same pattern as the number of the selected Stripes, and are less or close to 2% of the total cardinality of the striped dataset. Query a6 is again an exception as it requires a larger number of input Stripes, however the percentage of the required number of Stripe nodes is less than the corresponding percentage of Stripes, close to 39%, as nodes are not uniformly distributed in Stripes. On the contrary, queries c1 and c3 present a larger percentage of required Stripe nodes, close to 4.5%, for the same reasons. Regarding Stripe size, the same trend as the one concerning the actual number of Stripes required for query evaluation, is depicted in Figure 4.3(c). Note that the percentage of the size of the Stripes required for evaluating query a6 (22%) is even smaller than its corresponding percentage regarding the Stripe cardinality. This is due to the fact that the majority of Stripes selected for the evaluation of the "//"' step expression, concern Path Stripes that require less storage compared to Attribute or Data Stripes that store large textual data. All these observations are evident in Figure 4.4, where the normalised usage of Stripes is shown, along with the Stripe cardinality and size with respect to the striped dataset.

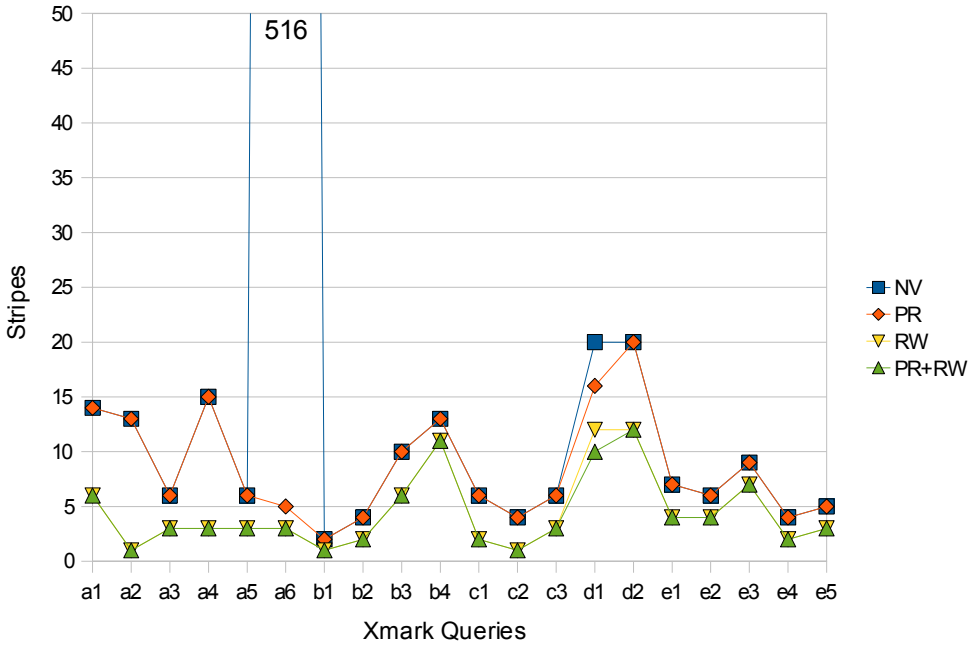


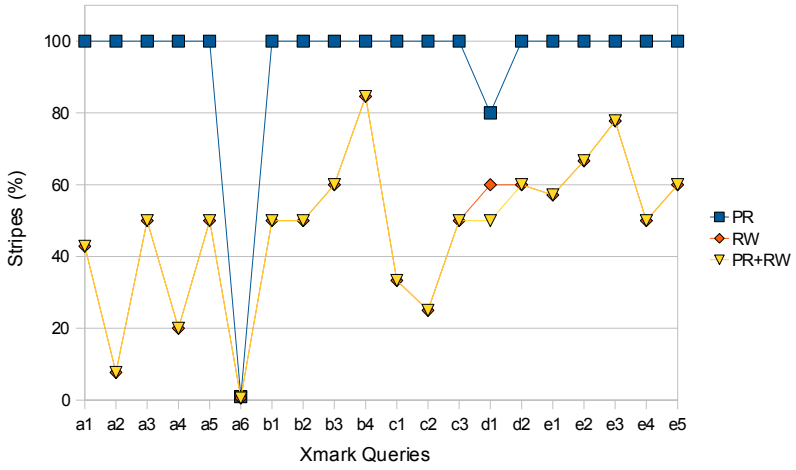
Figure 4.5: Impact of Pruning, Rewriting and their combination on query input for the Xmark dataset

4.5.1.2 Impact of Stripe Pruning

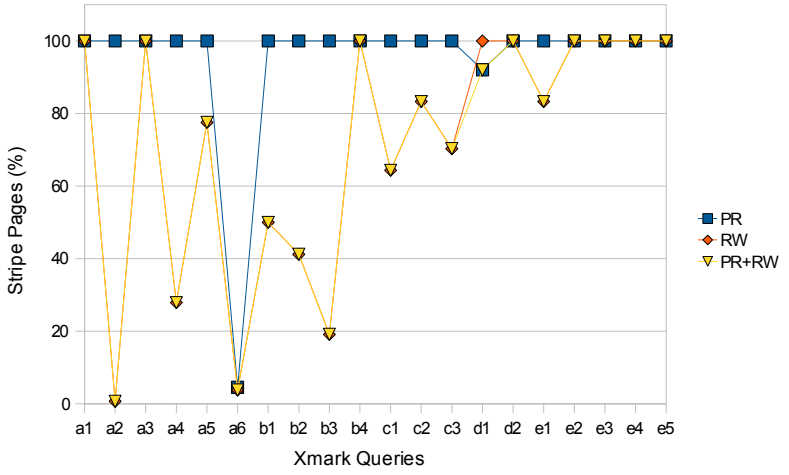
Both Pruning and Rewriting, may further reduce the number of selected Stripes for query evaluation. Their impact on query input for the Xmark query testbed is depicted in Figure 4.5. We also normalised the results of applying Pruning, Rewriting and both and the results as shown in Figure 4.6. We make the following observations:

Pruning The added impact of Pruning with respect to the naïve approach that merely uses Stripe Projection to select relevant to queries Stripes, is minimal for most of the tested queries. It only affects query a6 (by a large factor) and query d1. For query a6, the process prunes all Path Stripes initially selected by Stripe Projection for the "//" expression but do not contribute to final query results. For query d1, a small number of Stripes are pruned due to the value-based selection predicate, *i.e.,* Stripes that are guaranteed not to contain the desired values are removed. The reduction on number of selected Stripes for these queries is also reflected in the size of the selected Stripes, as depicted in Figure 4.6(b).

Rewriting The added impact of Rewriting with respect to the naïve approach, varies significantly. For queries b4, e2, e3, it has a small effect, achieving reduction



(a) Stripe reduction



(b) Stripe size reduction

Figure 4.6: Normalised impact of Pruning, Rewriting and their combination on query input for the Xmark dataset

on the number of Stripes up to 30% with respect to the naïve approach. On the other hand, query input for a2, a4, a6 and c2 is reduced by 75% up to 99%, while for the rest of the queries the reduction of the number of selected stripes ranges from 40% to 65% with respect to the naïve approach. However, when examining the reduction of Stripe size in Figure 4.6(b), we notice that it does not have the same effect as presented by the number of Stripes being reduced. In particular, Stripe size does not seem to be reduced at all for queries a1, a3, b4, d1-e5, although their corresponding number of Stripes is reduced for some of them by a large factor. This occurs because for those queries, the reduced

Stripes are mainly Stripes with a short path that only contain a few Stripe nodes and thus their size in insignificant compared to other Stripes that contain many nodes. The same applies for queries a5, c1-c3 for which although a reduction of Stripe size occurs, it is not proportional to the reduction of number of Stripes. On the other hand, for queries a2, a4, a6, b1, b2 the reduction of Stripe size is proportional to that of Stripes, while query b3 benefits from Rewriting more than the reduction of actual number of Stripes indicates.

Pruning and Rewriting The combined impact of applying Pruning and Rewriting is largely dominated by the impact of applying Rewriting alone. Only query d1 benefits from both processes, and only by a small factor as shown from the Stripe size reduction.

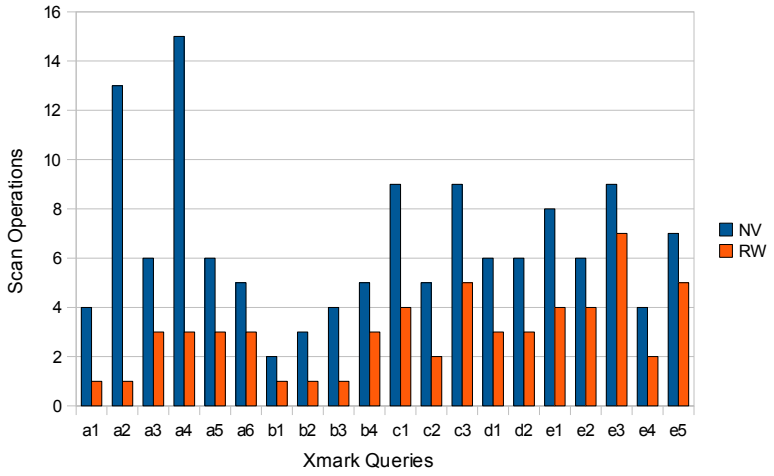
4.5.1.3 Impact of Operations Pruning

The impact of Rewriting in terms of operations reduction for the Xmark queries is depicted in Figure 4.7; Figure 4.7(a) presents the number of Scan operations compared to those that are performed in the naïve approach, *i.e.,* when Rewriting is not applied. For most cases, the Scan operation reduction is proportional to the reduced number of Stripes achieved by Rewriting. However, for certain queries the reduction of Scan operators deviates from the number of Stripes being pruned, due to the fact that multiple Stripes are merged in a single Merge operator. The reduction of Structural Join operations is presented in Figure 4.7(b); in many cases, a significant number of Structural Join operations is effectively discarded.

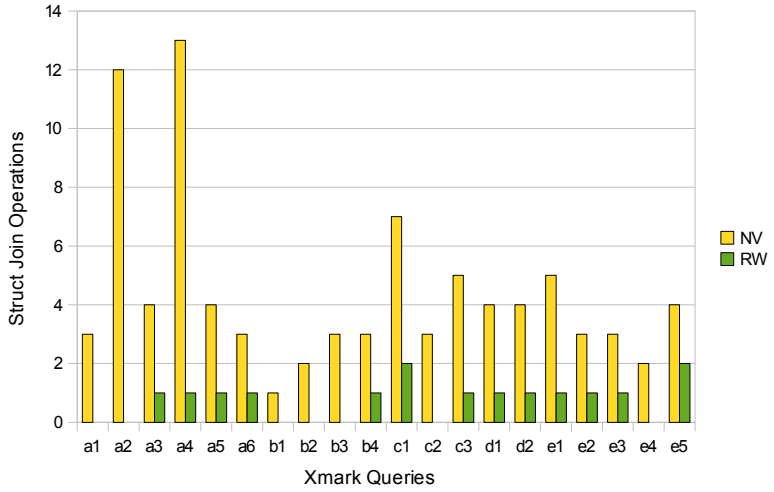
4.5.1.4 Query Performance

We now present the query evaluation performance of our native store prototype SRX for the Xmark dataset. We focus on the largest document produced for the Xmark dataset (sf=100, size=11GB); the evaluation times of all setups are shown in Table 4.3. We highlight the best time of all considered setups for each of the tested queries.

We first focus on the impact of applying each optimisation in isolation. The results are presented in Figure 4.8. Regarding the PR setup, we observe that the evaluation times are almost the same as the ones of the NV setup for most of the tested queries. The only exception is query a6 for which Pruning achieves significant reduction both in number of Stripes and Stripe size. Pruning also has a small impact on the response time



(a) Scan operations reduction



(b) Structural join operations reduction

Figure 4.7: Impact of Rewriting with respect to operators reduction for the Xmark dataset

of query d1 due to the reduction of its input size by a small factor. An interesting point is that for some queries, the PR setup requires slightly more time than the NV setup. This is because of the (relatively small) added cost of applying the Stripe Pruning process, combined with the fact that it actually has no effect since no further Stripes are pruned.

Regarding the OP setup, the evaluation times are almost the same as the ones of the NV setup for most of the tested queries. The only exception is query d1 for which the Stripe-aware Scan operators access fewer Stripe nodes by a factor of 66% with respect to the NV setup. On the other hand, we observe that for some queries the evaluation

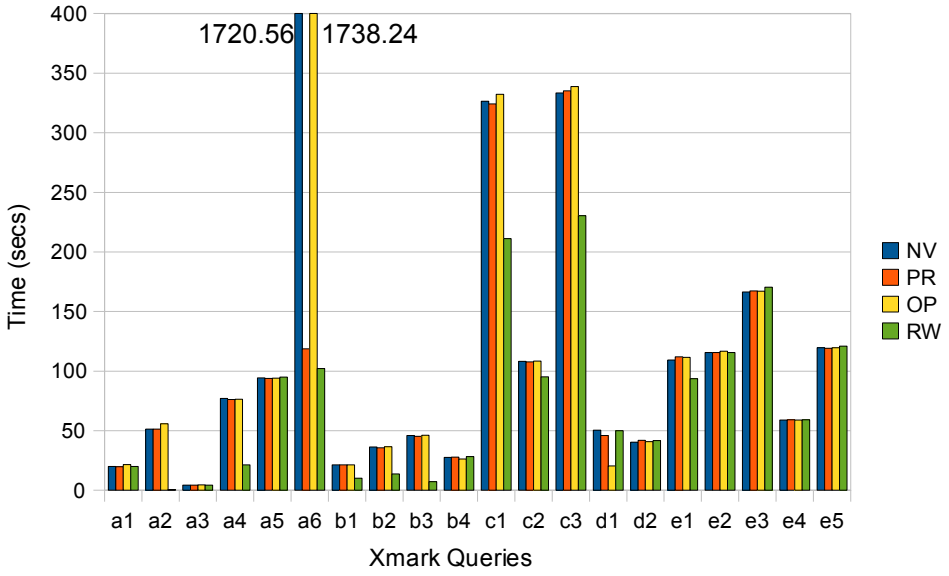


Figure 4.8: SRX query evaluation results for each optimisation in isolation

time of the OP setup is slightly bigger than that of the naïve setup. This is due to the minimal impact of Stripe-aware Optimisation on the input Stripes, in addition to the fact that certain Stripe-aware algorithms (for instance, the level-based Scan operators) are computationally more expensive than their Stripe-unaware counterparts.

For the RW setup, we first observe that it has a significant impact on query evaluation in comparison to the PR and OP setups. With respect to the NV setup, we observe that the impact of Rewriting on evaluation times reflects its impact on the input size and is almost immune to its impact on the reduced evaluation operations. This is clearly shown in Figure 4.9, where the normalised effect of Rewriting process is displayed for (a) Stripe size, (b) operations, and (c) evaluation time with respect to the NV setup; the speedup on evaluation times for all Xmark queries is proportional to the reduction of Stripe size (in I/O pages) and completely immune to the reduction of the total number of operations caused by Rewriting. An interesting observation is that only query a5 does not follow such a pattern. This is because for this specific query there exists a very selective value-based selection that dominates the evaluation time; the Stripe size reduction that is performed by the Rewriting process will have an impact on the evaluation time only if all Stripe contents are retrieved. The existence of the value predicate prevents the Stripe contents (reduced by the RW setup) being accessed, even for the NV setup and thus the input reduction that occurs in the RW setup has practically no impact

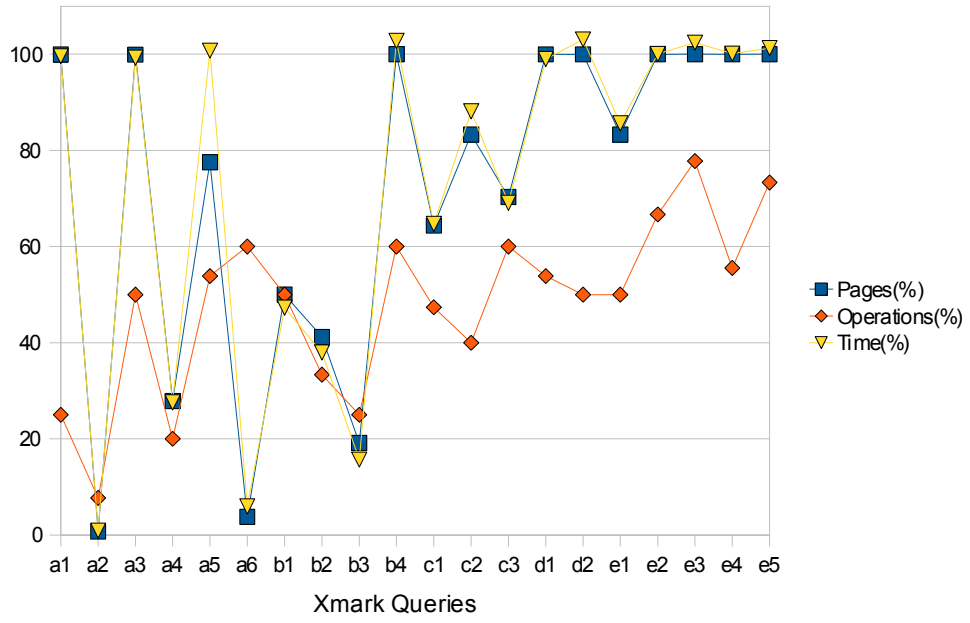


Figure 4.9: Normalised reduction on Stripe storage, operations and evaluation time for RW setup with respect to NV setup for Xmark queries

on the evaluation time.

So far, we have seen that for most Xmark queries, one of the three optimisations discussed can impact the query evaluation time, with Rewriting being the one standing out. When combining optimisations, the evaluation time of the combined setup is usually determined by the dominant optimisation, *i.e.,* the optimisation that when applied in isolation, has the biggest impact on query evaluation time. For instance, the evaluation time for query a6 at PR+OP setup, is dominated by the impact of Pruning, while the evaluation time for query d1 is dominated by the impact of applying Stripe-aware Optimisation. Similarly, for the RW+PR setup, evaluation times are influenced by the impact of Rewriting, which always suppresses the impact of Pruning for Xmark queries. The same observations apply for the RW+OP setup, where Rewriting has the dominant impact on query evaluation times, with the exception of query d1, where the combined setup gains from the optimised Stripe-aware algorithms. Finally, the PRO setup, as expected, is guaranteed to benefit from each of the optimisations applied being dominant. For Xmark queries, the PRO setup mainly benefits from Rewriting optimisation. An interesting point is that the PRO setup does not provide the best evaluation time in most cases for Xmark queries. However, its results deviate from the best evaluation result by less than 1% for all queries, but two; query a3 which deviates by

Xmark100 (11 GB)								
Query	NV	PR	OP	RW	PR+OP	RW+PR	RW+OP	PRO
a1	19.99	**19.81**	21.53	19.9	19.88	19.95	19.82	19.95
a2	51.32	51.21	55.89	0.48	51.45	**0.46**	**0.46**	**0.46**
a3	4.42	**4.39**	4.53	**4.39**	4.43	4.45	4.45	4.44
a4	77.01	76.22	76.29	21.23	75.71	21.3	21.04	**20.98**
a5	94.32	**93.89**	94.05	95.02	94.99	94.71	94.09	94.1
a6	1720.56	118.65	1738.24	**102.12**	118.93	102.28	103.1	102.26
b1	21.39	21.26	21.31	**10.11**	21.28	10.12	10.22	10.13
b2	36.35	35.74	36.67	13.79	36.64	13.75	**13.67**	13.77
b3	45.83	45.37	46.25	**7.18**	45.62	7.25	**7.18**	7.25
b4	27.56	27.8	26.34	28.33	26.52	27.61	26.64	**26.27**
c1	326.45	324.19	332.33	211.21	326.93	211.23	211.43	**210.8**
c2	108.1	107.67	108.33	**95.25**	108.11	**95.25**	95.61	95.43
c3	333.26	335.09	338.83	230.31	332.31	228.84	**227.39**	227.62
d1	50.4	45.85	20.32	49.89	**20.13**	47.49	20.14	20.24
d2	**40.33**	42.01	40.74	41.57	41.88	41.38	41.17	40.68
e1	109.31	111.9	111.55	**93.58**	110.45	94.71	94.71	94.1
e2	115.58	115.55	116.62	115.65	115.89	115.86	**115.05**	115.1
e3	166.43	167.26	166.98	170.45	**165.11**	170.35	167.89	171.65
e4	59.01	59.16	59.03	59.09	58.85	59.07	58.75	**58.32**
e5	119.51	**119.13**	119.52	121.01	120	119.56	120.38	119.39

Table 4.3: SRX query performance for Xmark queries (Xmark100)

1.1% and e3 which presents the maximum deviation: 3.8%. The results for all tested optimisation setups running on the largest Xmark dataset considered (Xmark100) are shown in Figure 4.10. The same observations hold for the rest of the Xmark datasets, ranging from 11MB to 1.1GB; the results are presented in Figure 4.11.

4.5.2 Mbench

We now conduct the same experimental study for the Mbench datasets.

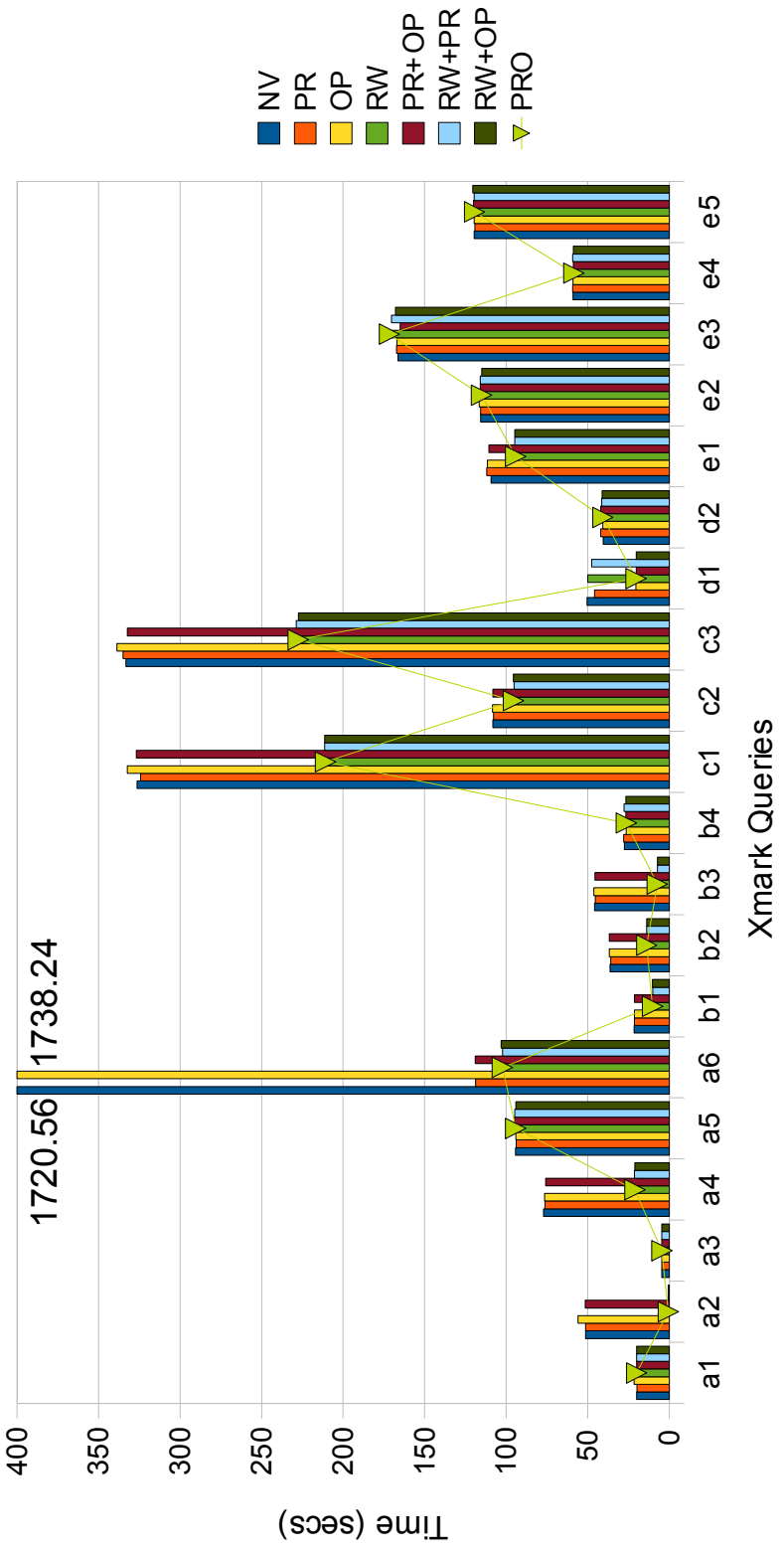
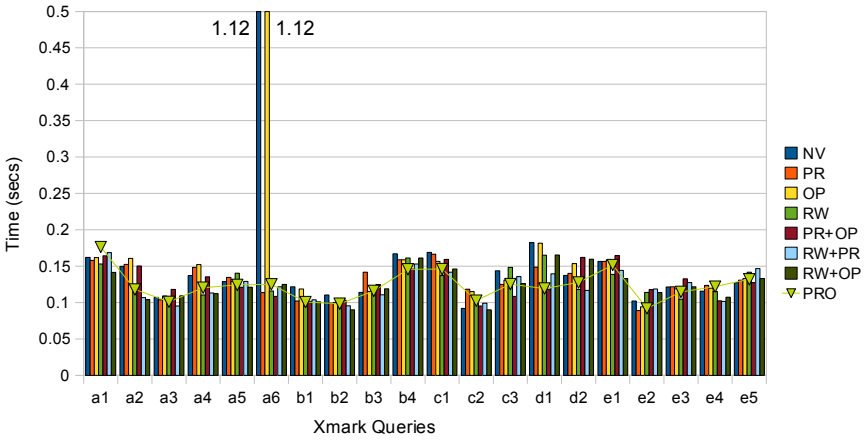
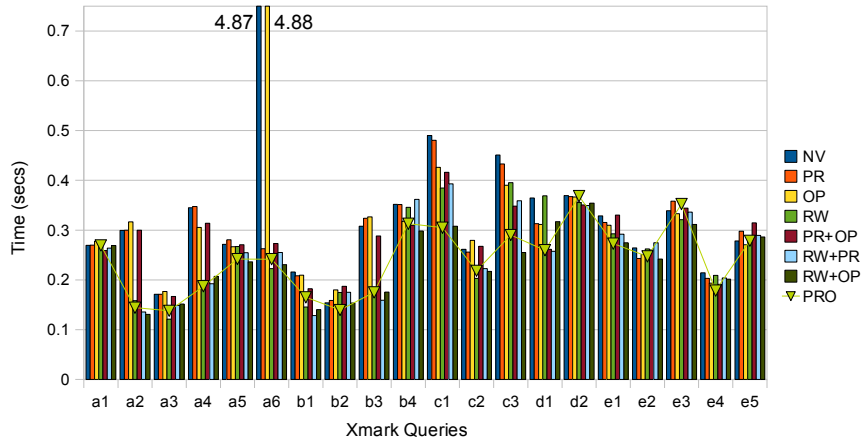


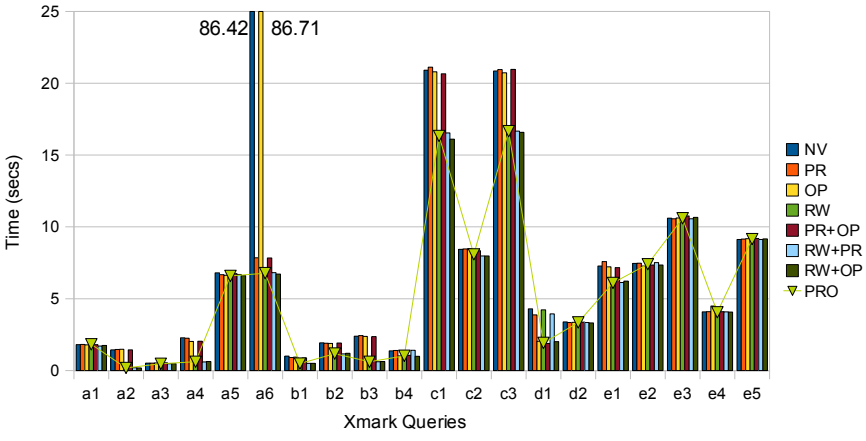
Figure 4.10: SRX query evaluation for the Xmark100 dataset



(a) Xmark0.1



(b) Xmark1



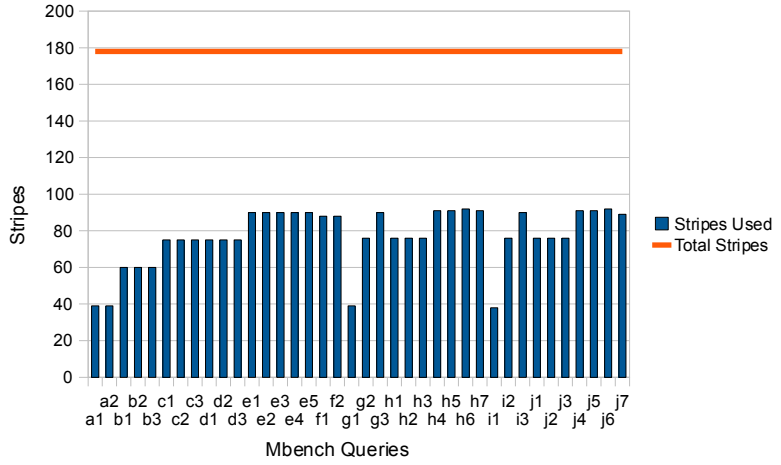
(c) Xmark10

Figure 4.11: SRX query evaluation for the Xmark0.1, Xmark1 and Xmark10 datasets

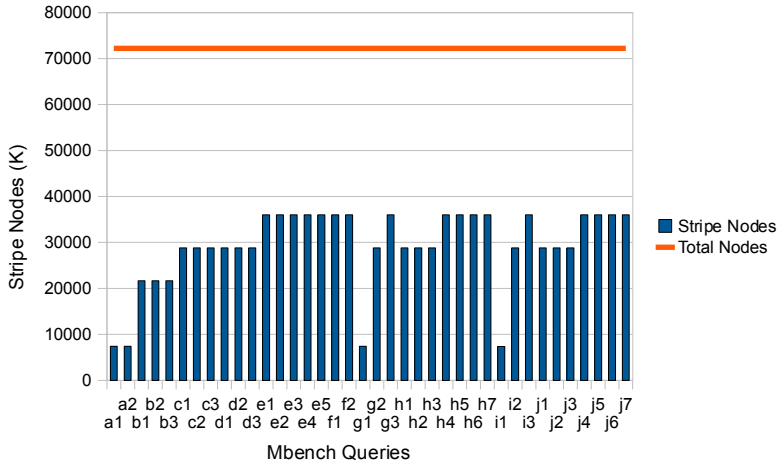
4.5.2.1 Impact of Striping

The number of the selected Stripes for the Mbench queries are presented in Figure 4.12. Striping has a profound impact on Stripe selection for all Mbench queries, narrowing down the number of Stripes by 57% on average with respect to the total number of Stripes of the striped dataset. Note that the reduction of the number of Stripes achieved for the Mbench dataset is not at the same level as the one achieved for the Xmark dataset. This is due to the Mbench document structure, having a small number of unique tag names and being highly recursive. However, even in such a case, Striping effectively discards all Stripes that are not involved in the queries tested, achieving satisfactory reduction in terms of number of Stripes.

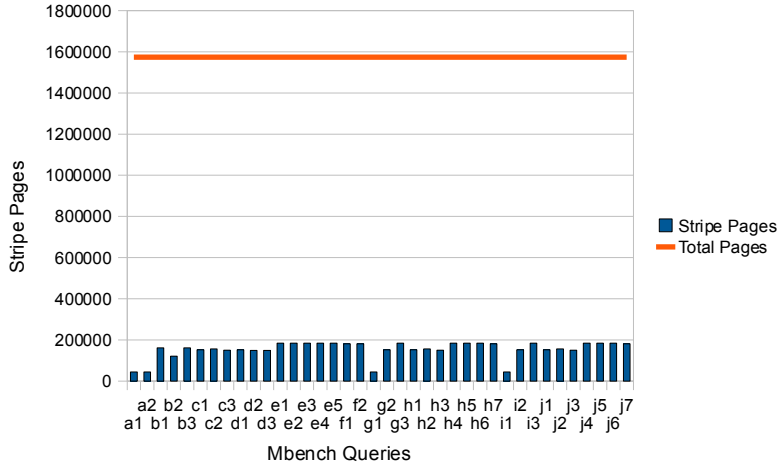
In addition to the actual number of Stripes selected each time for query evaluation, we also measured the impact of Striping in Stripe cardinality and size. The results are depicted in Figures 4.12(b) and 4.12(c) respectively. The reduction of Stripe cardinality is for most cases proportional to the reduction of the number of Stripes, with an average of 60% reduction of the number of Stripe nodes. However, for some queries (a1, a2, g1, i1), the effect of Striping has greater impact on Stripe cardinality. Regarding the impact of Striping in terms of Stripe size, Figure 4.12(c) indicates that the size of the Stripes that are selected for query evaluation in only the 10% in average of the size of the striped dataset and thus, the average reduction percentage in terms of number of Stripe pages, reaches 90%, much higher than the equivalent average reduction percentage for the number of Stripes (57%) and the number of Stripe nodes (60%). This is evident in Figure 4.13, which depicts the normalised usage of Stripes along with the Stripe cardinality and size with respect to the striped dataset. The deviation of these values is a side-effect of the type of Stripes that are selected for most Mbench queries. The Mbench dataset is a highly recursive dataset that contains element nodes with long attribute lists and long textual content. Mbench queries mostly involve structural join operations among elements of varying selectivities that are determined by value-based predicates on the element attributes. Our Striping decomposition enforces the separation of structure from content. Thus, for Mbench queries, Striping effectively prunes Value Stripes that are not needed. As a result, the size of the selected Stripes is only a small portion of the overall size of the striped document. since as seen in Table 4.2, the size of Data Stripes dominates the total size of the striped Mbench dataset.



(a) Usage of Stripes



(b) Cardinality of the selected Stripes



(c) Size of the selected Stripes

Figure 4.12: Impact of Striping on the Mbench dataset

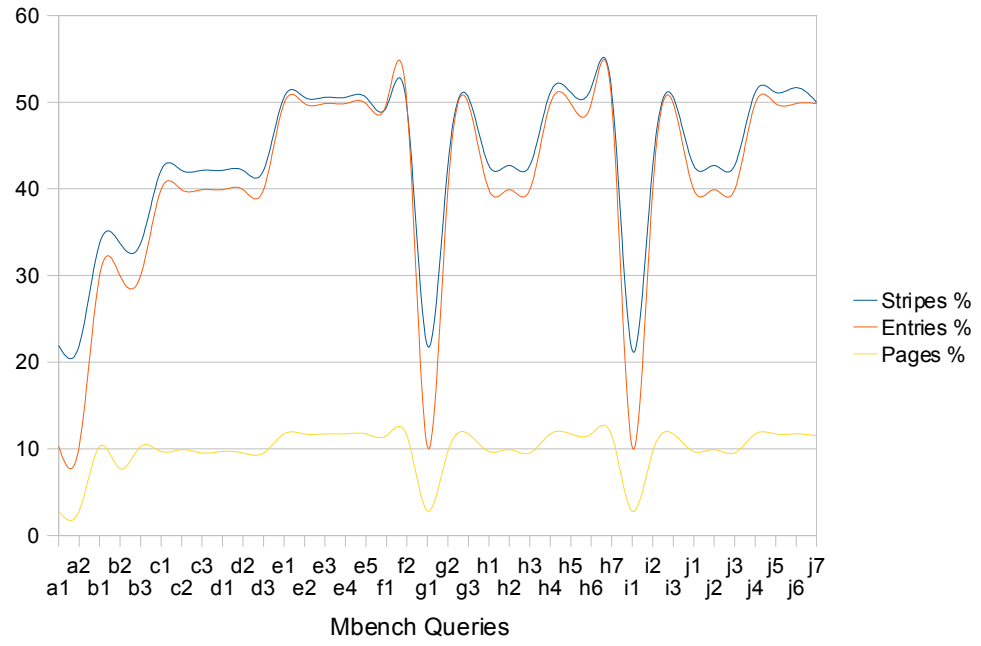


Figure 4.13: Normalised impact of Striping on the Mbench dataset

4.5.2.2 Impact of Stripe Pruning

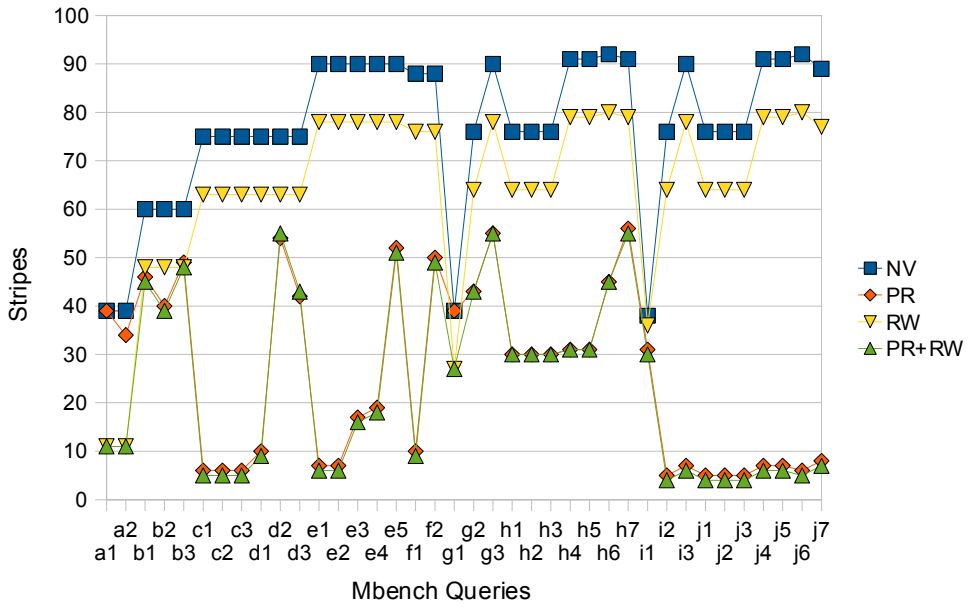
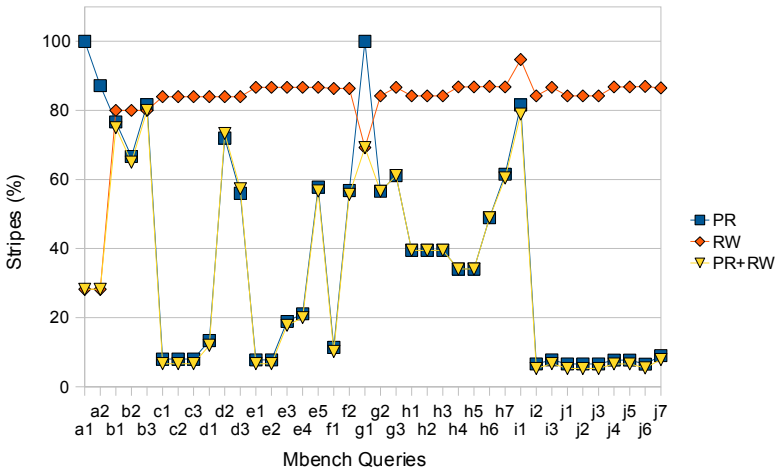
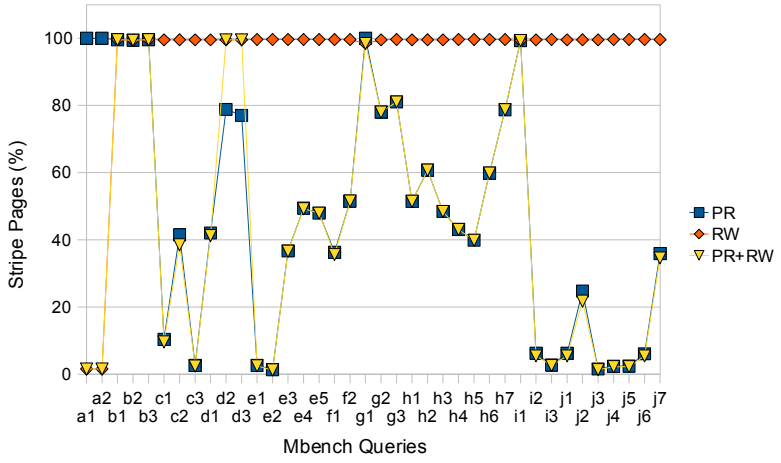


Figure 4.14: Impact of Pruning, Rewriting and their combination on query input for the Mbench dataset



(a) Stripe reduction



(b) Stripe size reduction

Figure 4.15: Normalised impact of Pruning, Rewriting and their combination on query input for the Mbench dataset

The impact of Pruning and Rewriting on query input for the Mbench query testbed is shown in Figure 4.14. We also normalised the results of applying Pruning, Rewriting and both and the results are shown in Figure 4.15. We make the following observations:

Pruning Unlike the Xmark dataset, the added impact of Pruning with respect to the naïve approach further reduces the number of the selected Stripes by 62% on average for all Mbench queries. In particular, for only 5 queries, the reduction of number of Stripes is below 20%, while for 17 queries it is above 80%. However, the reduction on the number of Stripes for these queries is not always reflected in the size reduction of the Stripes that are used during query evaluation, as de-

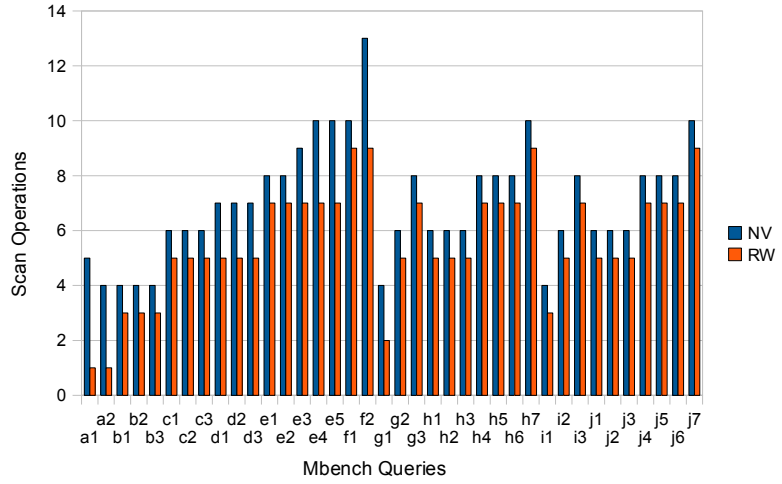
picted in Figure 4.15(b). In particular, queries a2, b1, b2, c2, e3, g2-h2, i1, j2 and j7 benefit by a smaller factor by Pruning than what the actual number of Stripes being pruned indicates. This happens because for those queries, the pruned Stripes contain a small number of nodes compared to those that are actually accessed in order to evaluate such queries. Nevertheless, Pruning achieves to reduce the size of the required Stripes by 50% on average for all queries, a significant result by all means. On the contrary, queries c3, e1, e2, i3 and j3-j5 benefit by a larger factor compared to that considering the number of Stripes being reduced. In these cases, the pruned Stripes contain noticeable amounts of nodes and this is conceived as bigger I/O savings.

Rewriting The added impact of Rewriting for the Mbench dataset with respect to the naïve approach, does not have the same effect as it has for the Xmark dataset. This is depicted both in Figures 4.14 and 4.15(a), where it is shown that Rewriting manages to prune a small number of Stripes. Furthermore, as shown in Figure 4.15(b), the reduction of such Stripes has minimal impact to Stripe size reduction, as the pruned Stripes only contain a small fraction of nodes compared to the Stripes needed for query evaluation. The average reduction of the number of Stripes caused by Rewriting is less than 20%, while the average reduction of Stripe size is only close to 5%. The only exceptions that Rewriting achieves a significant reduction both for the number of Stripes and Stripe size are queries a1 and a2, as these are rewritten to shorter expressions that require minimum input.

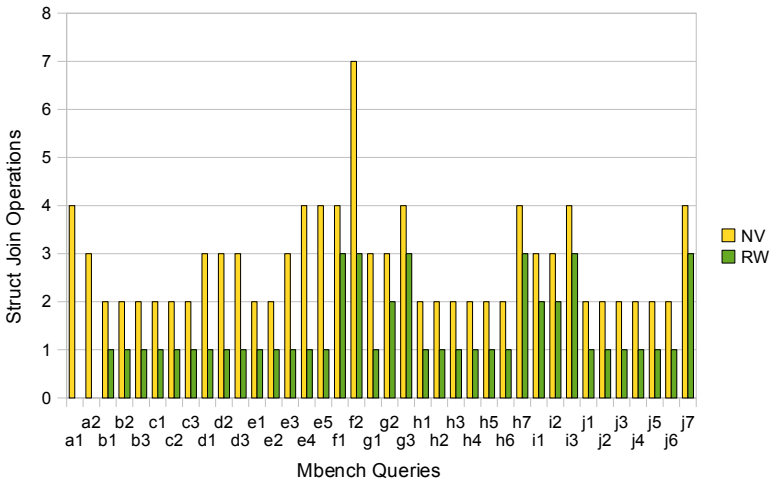
Pruning and Rewriting The combined impact of applying Pruning and Rewriting is largely dominated by the impact of applying Pruning alone, since Rewriting has practically no added effect compared to the naïve approach.

4.5.2.3 Impact of Operations Pruning

We now report the impact of Rewriting in terms of operations reduction. The results are depicted in Figure 4.16. The average Scan operation reduction is proportional to the average reduction of the number of Stripes achieved by Rewriting, close to 20%. However, the average reduction of Structural Join operations is close to 50%, meaning that half of the structural join operations on average are removed from Rewriting. Nevertheless, our experience from the Xmark dataset is that although this reduction in operations is substantial, it will not be reflected in response times, as the I/O cost dom-



(a) Scan operations reduction



(b) Structural join operations reduction

Figure 4.16: Impact of Rewriting with respect to operators reduction for the Mbench dataset

inates the query evaluation cost and we already presented that the Stripe size reduction from applying Rewriting for Mbench queries is minimal.

4.5.2.4 Query Performance

We now present the query evaluation performance of our native store prototype SRX for the Mbench dataset. We focus on the largest document produced for the Mbench dataset (sf=10, size=5GB). The evaluation times for all possible evaluation setups for the Mbench10 dataset are shown in Table 4.4. We highlight the best time of all considered setups for each of the tested queries.

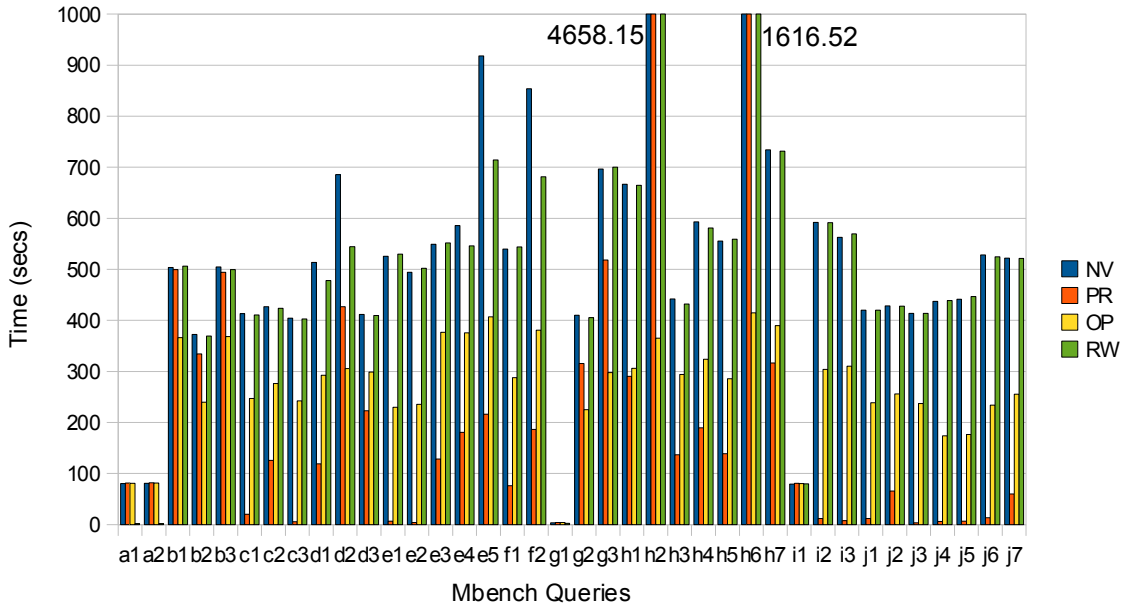


Figure 4.17: SRX query evaluation results for each optimisation in isolation

We begin our analysis focusing on the impact of applying each of the optimisations in isolation. To that end, we gathered the results for NV, PR, RW and OP evaluation setups and we present them in Figure 4.17.

Regarding the PR evaluation setup, we observe that the evaluation times are substantially better compared to the ones produced by the NV setup; the PR setup outperforms the NV setup by almost 60% on average, while only 6 queries remain unaffected by it. Note that the time speedup for the PR setup is proportional to the reduction of the number of input Stripes and close to the Stripe size reduction, which mainly explains the reason that the PR setup outperforms NV. For some of the queries that are unaffected by applying Pruning, the evaluation time of the PR setup is slightly greater than the one of the NV setup, *e.g.,* queries a1 and i1. This is because of the (relatively small) added cost of applying the Pruning process, combined with the fact that it actually has no effect since no further Stripes are pruned.

Similar results are observed when considering the OP evaluation setup, where we observe a speedup by 40% on average compared to the NV setup. This speedup is the result of using optimised Stripe-aware algorithms in combination with the pipelined evaluation of filter expressions. Our filter evaluation algorithms, described in Section 3.3.3, although they do not perform a single scan over their inputs, they avoid

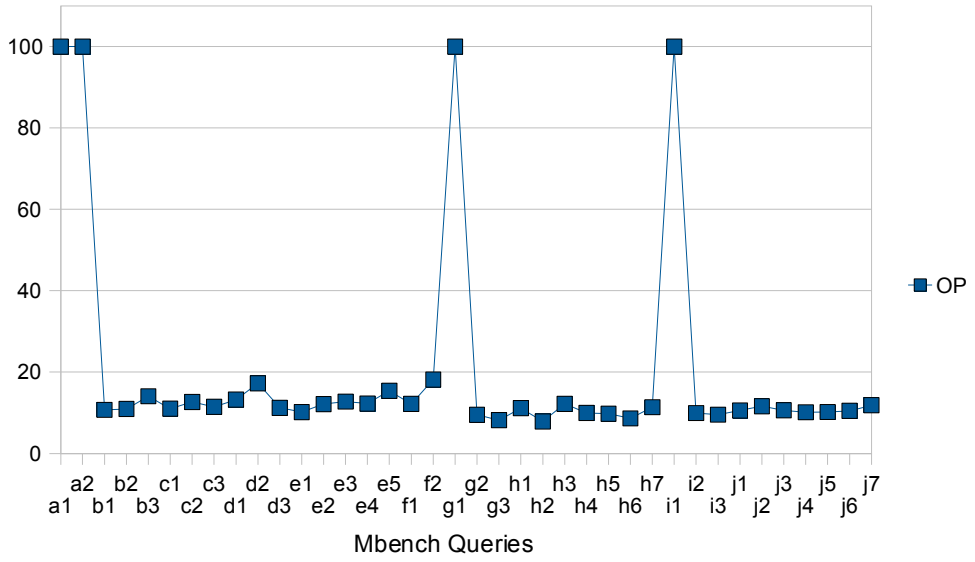


Figure 4.18: Impact of applying Stripe-aware Optimisation; normalised Stripe nodes access with respect to the naïve setup for the Mbench dataset

producing large intermediate results, and thus are evaluated on the fly for each context node. Filter evaluation algorithms also avoid full result computation of the predicate expression; as soon as the first result is computed that satisfies the filter predicate, the evaluation continues with the next context node. This pipelined evaluation mode favours queries over the Mbench dataset which is deeply-nested and highly recursive. Furthermore, Stripe-aware filter evaluation processing, described in Section 3.4.2, further optimises the evaluation of parent-child and sibling location steps that occur in filter expressions; this is done by employing specialised, level-based Scan operators that reduce the search space of candidate nodes with respect to a given context node and thus access fewer Stripe nodes compared to the algorithms used for the NV setup. We compared the numbers of Stripe nodes that are retrieved when using both the NV and OP setups and observed that when the Stripe-aware algorithms are used the total number of Stripe nodes is reduced by almost 80% on average. Queries a1, a2, g1 and i1, where the results for both setups are the same, do not contain any filter expression; indeed we verified that the number of accessed nodes is the same for both evaluation setups. These are depicted in Figure 4.18.

For the RW evaluation setup, we observe that as expected, it has minimal impact on query evaluation times for the Mbench queries compared to the NV setup. For most of the queries, the evaluation times are comparable to the ones of the NV setup; an expected result due to the minimal impact of Rewriting on Stripe size reduction

as already described in Section 4.5.2.2. The only exceptions are queries a1, a2 for which the RW evaluation setup performs better than any other setup described so far due to the maximal Stripe size reduction it achieves, combined with the elimination of all structural join operations. Furthermore, noticeable speedup is achieved for queries d2, e5 and f2 that use extensively "//" operators and for which the RW setup has a significant impact on structural join operation reduction. All other queries are immune to Rewriting; even if certain structural join and Scan operations are reduced, this is not reflected in the response times.

We now conclude our analysis with the combined evaluation setups, *i.e.,* setups that combine any two or all optimisations. We described that for most Mbench queries, one of the three optimisations largely affects query evaluation time; this time Pruning is the one that has the greatest impact on evaluation times, as opposed to Rewriting that had a similar effect for the Xmark dataset. When combining optimisations, the evaluation time of the combined setup is usually determined by the dominant optimisation, *i.e.,* the optimisation that when applied in isolation, has the biggest impact on query evaluation time. However, for some queries, combined setups present the added impact of their optimisations.

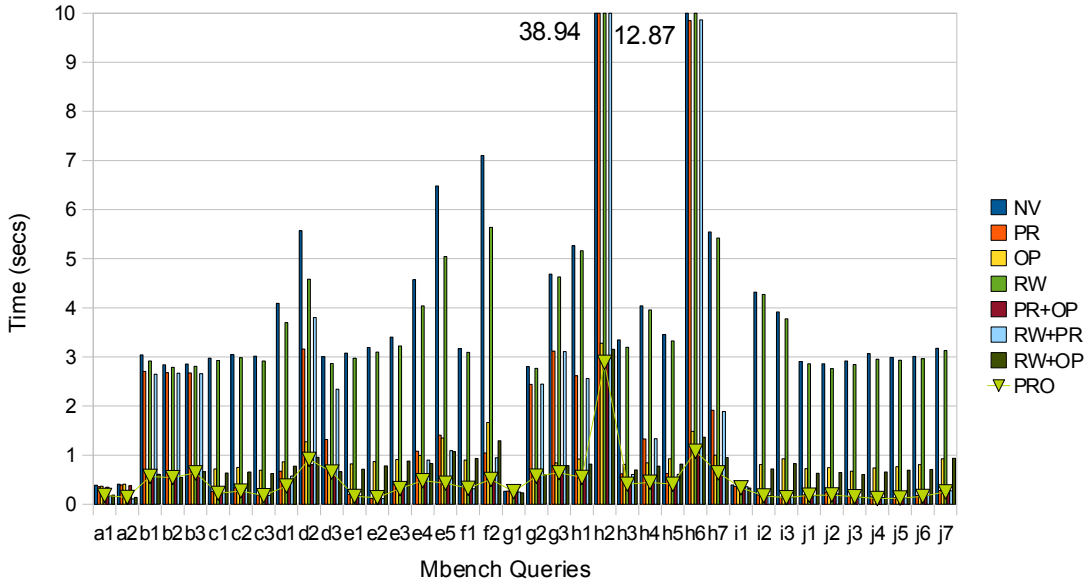
For evaluation setup PR+OP, the evaluation times are usually dominated by Pruning, which has the largest impact on most of the queries. However, for queries b1-b3 and g2, the evaluation time is dominated by employing Stripe-aware evaluation algorithms. Finally, for queries d2, d3, e5, g3, h1, h2, h4, h6 and h7, the PR+OP evaluation setup performs better than any of the setups where optimisations are applied in isolation, gaining from their combined effect on Stripe size reduction and the reduced Stripe node access. On the other hand, combined evaluation setups RW+PR and RW+OP, are mostly dominated be either Pruning in the former case or Stripe-aware Optimisation in the latter, since with the exception of queries a1 and a2, Rewriting has minimal effect on Mbench queries. Finally, for the PRO evaluation setup, which combines all optimisations, query evaluation times are affected by the optimisation that is dominant for each query, while in most cases, evaluation times are improved by the combined effect of all optimisations. The results for all tested optimisation setups running on the largest Mbench dataset considered (Mbench10) are shown in Figure 4.20. Similar observations hold for the rest of the Mbench datasets, ranging from 50MB to 500MB and their evaluation results are presented in Figure 4.19.

Query	Mbench10 (5 GB)							
	NV	PR	OP	RW	PR+OP	RW+PR	RW+OP	PRO
a1	80.27	81.34	80.94	1.84	82.08	1.89	1.91	1.91
a2	80.68	81.63	81.41	1.78	82.14	1.81	2.06	**1.78**
b1	503.77	499.35	366.25	506.01	366.02	502.6	361.25	**354.29**
b2	372.64	334.43	239.77	369.15	240.73	332.45	**235.09**	238.59
b3	504.51	494.35	368.07	499.42	370.86	501.15	359.18	**358.7**
c1	413.09	20.07	247.25	410.43	19.64	18.63	239.7	**18.3**
c2	427	125.5	276.43	424	125.58	119.4	273.17	**118.37**
c3	404.04	5.52	242.19	402.88	5.4	5.02	235.41	**5.01**
d1	513.42	119.11	292.57	478.06	117.03	116.53	284	**112.6**
d2	685.51	427.03	305.74	544.52	**226.8**	505.67	299.34	296.98
d3	411.48	223.03	298.76	409.63	**210.51**	330.89	293.28	292.17
e1	525.53	6.4	229.52	529.86	6.57	6.15	228.38	**6.04**
e2	494.28	3.75	235.32	502.02	3.83	**3.56**	230.96	3.57
e3	549.15	128.37	376.52	551.62	128.23	128.7	372.78	**127.4**
e4	585.89	180.45	375.34	546.01	174.96	178.85	381.99	**174.39**
e5	918.18	215.86	407.08	714.07	166.4	184.52	403.06	**165.26**
f1	539.95	76.44	287.9	543.64	76.01	75.2	285.06	**74.92**
f2	854.06	186.17	380.79	681.18	172.73	186.53	370.33	**171.53**
g1	3.61	3.79	3.82	2.35	3.84	2.36	**2.31**	2.34
g2	410.02	315.56	225.32	405.61	223.89	315.6	**217.96**	221.93
g3	696.62	518.51	297.56	700.39	268.25	518.03	289.21	**266.05**
h1	666.93	290.63	306.18	664.35	145.79	293.26	291.71	**141.5**
h2	4658.15	3964.34	364.97	4645.56	266.77	3998.73	353.05	**264.09**
h3	442.1	136.45	294.31	432.11	131.37	133.72	279.59	**129.62**
h4	592.93	189.38	324.05	580.86	151.44	187.9	317.77	**151.16**
h5	555.17	138.81	285.62	559.02	132.62	135.39	278.68	**131.41**
h6	1616.52	1136.4	414.97	1619.53	218.69	1136.08	391.54	**215.88**
h7	734.24	316.51	389.75	731.39	**248.47**	321.65	372.43	250.55
i1	79	80.88	80	79.81	80.28	80.04	**78.15**	79.65
i2	591.76	11.59	304	591.49	11.49	**10.08**	293.14	10.19
i3	562.9	7.51	309.98	569.46	7.64	6.88	296.44	**6.7**
j1	419.74	11.58	238.43	420.08	11.48	10.1	231.22	**10.06**
j2	428.53	65.44	255.83	427.85	66.11	57.98	247.01	**57.87**
j3	413.5	3.45	236.92	413.81	3.43	**2.97**	230.43	2.99
j4	437.41	6.32	173.72	438.52	6.35	6.07	168.01	**5.92**
j5	441.44	6.43	176.42	446.91	6.46	5.9	172.53	**5.85**
j6	528.4	13.54	233.74	524.31	13.56	**12.02**	226.34	12.06
j7	521.96	60.15	255.24	521.47	60.66	57.42	248.45	**56.09**

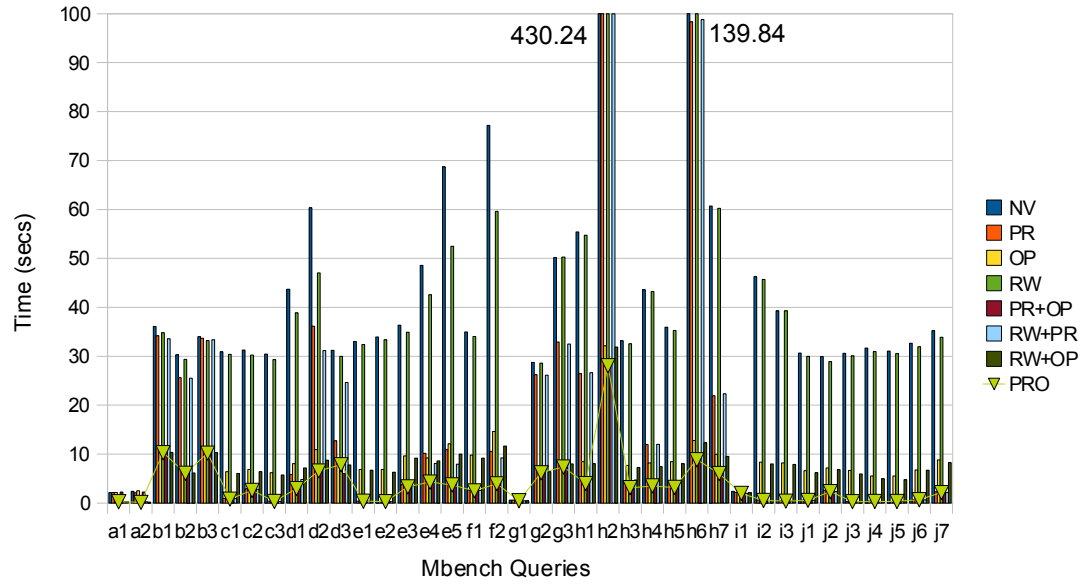
Table 4.4: SRX performance in Mbench queries

4.5.3 DBLP

We now conduct the same experimental study for the DBLP datasets.



(a) Mbench0.1



(b) Mbench1

Figure 4.19: SRX query evaluation for the Mbench0.1 and Mbench1 datasets

4.5.3.1 Impact of Striping

The number of the selected Stripes for the DBLP queries are presented in Figure 4.21. Striping has a profound impact on Stripe selection for all DBLP queries, reducing the number of Stripes by 73% on average with respect to the total number of Stripes of the striped dataset. For almost half of the DBLP tested queries, the achieved reduction in

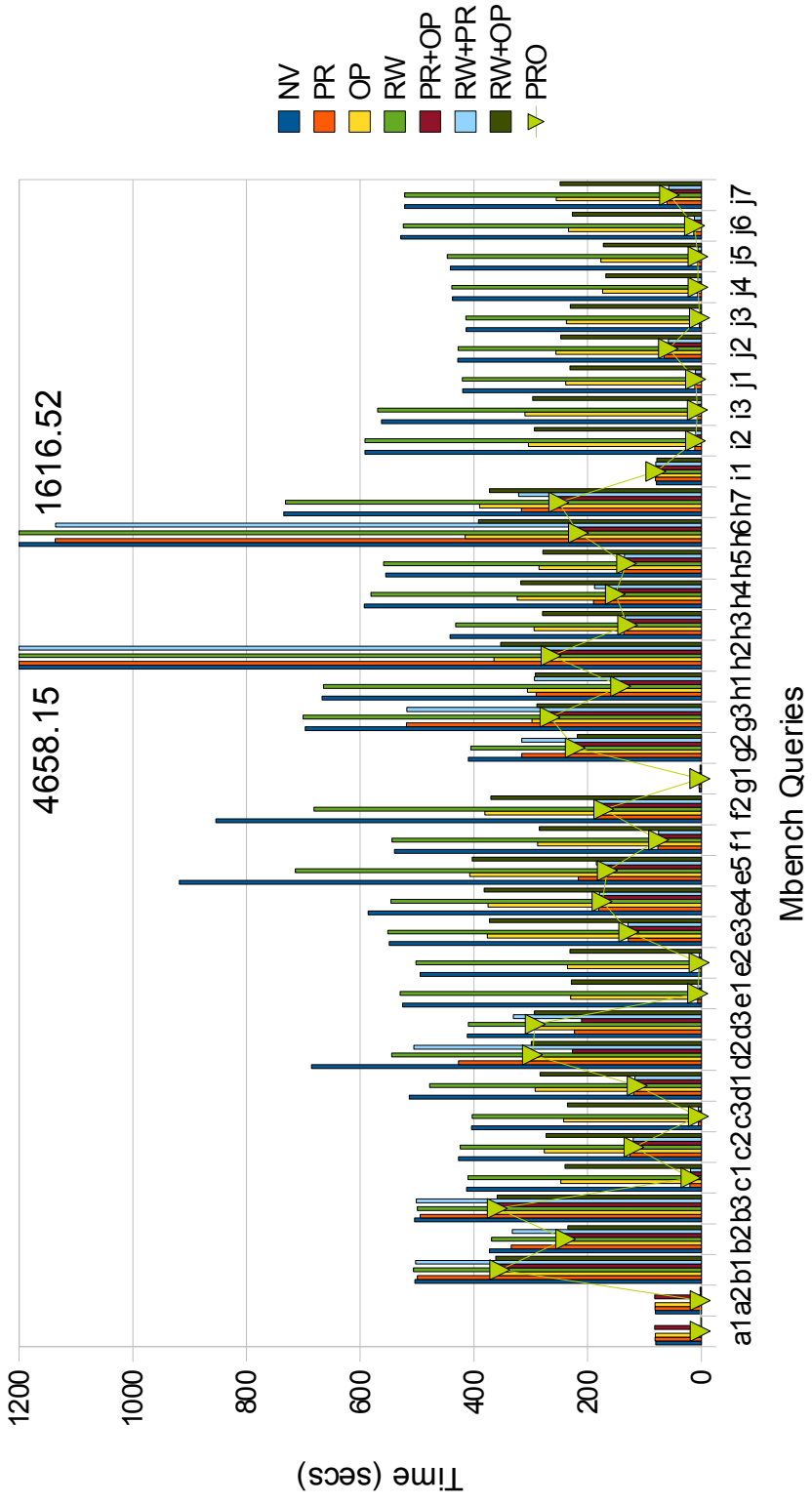
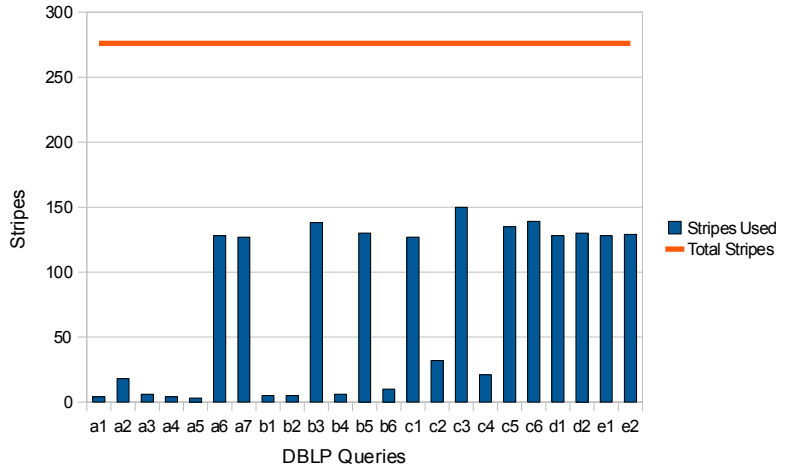
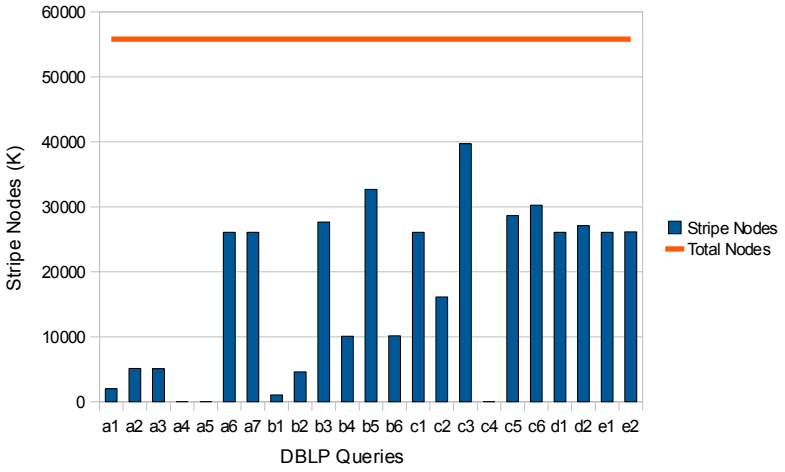


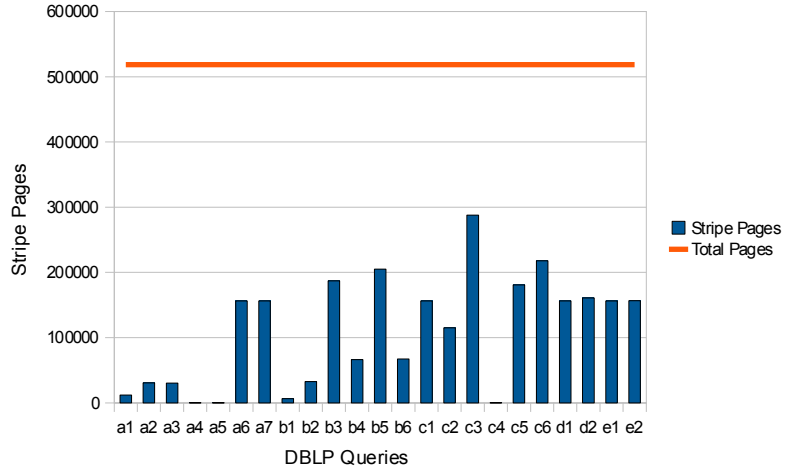
Figure 4.20: SRX query evaluation for the Mbench10 dataset



(a) Usage of Stripes



(b) Cardinality of the selected Stripes



(c) Size of the selected Stripes

Figure 4.21: Impact of Striping on the DBLP dataset

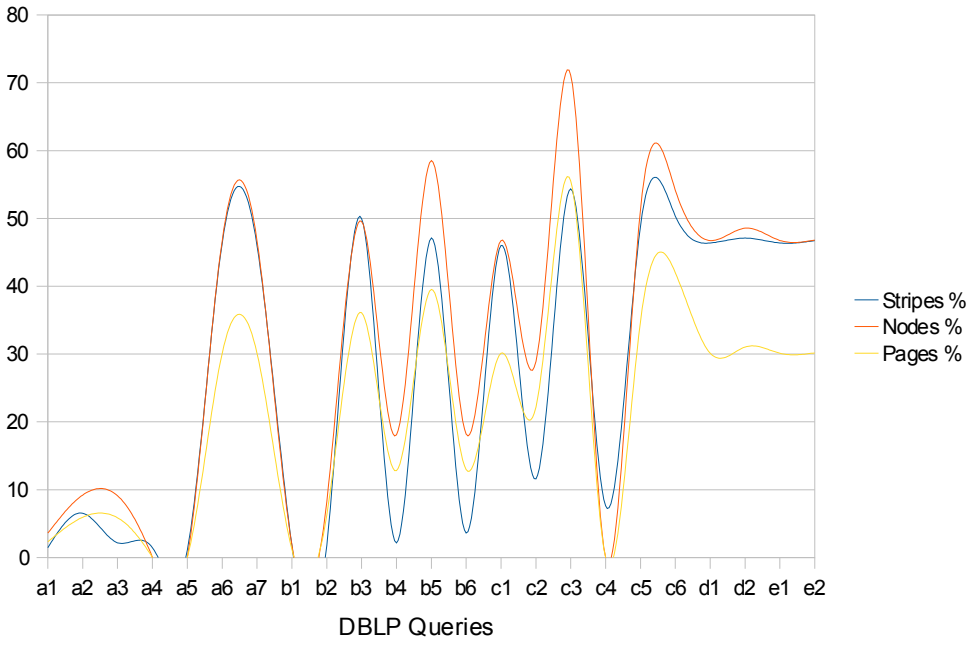
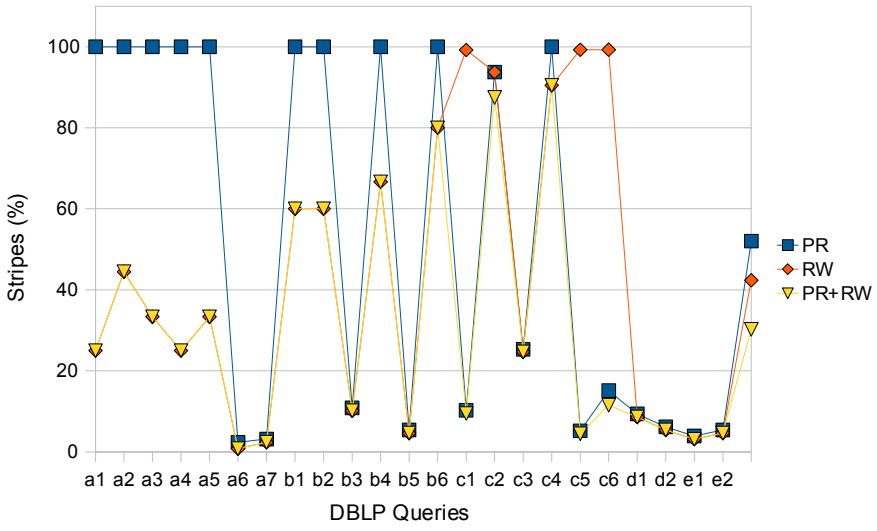


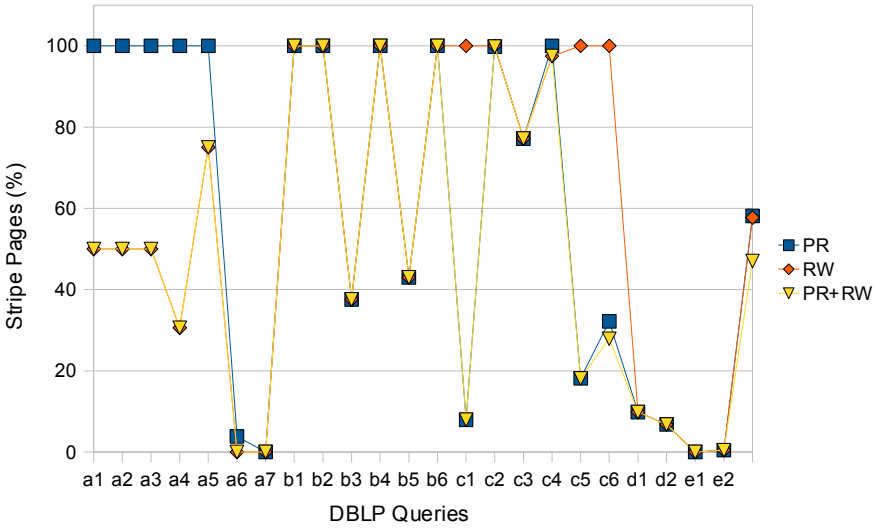
Figure 4.22: Normalised impact of Striping on the DBLP dataset

the number of Stripes is more than 90%, while for the rest of the queries, the number of the discarded Stripes reaches at least the 50% of the total number of Stripes of the striped dataset. Overall, Striping achieves satisfactory reduction in terms of the number of Stripes being selected for query evaluation.

As far as the impact of Striping on Stripe cardinality and size is concerned, the results for the DBLP dataset are shown in Figure 4.21(b) and Figure 4.21(c) respectively. The achieved reduction in Stripe cardinality is 70% on average and for most cases, it is proportional to the reduction in the number of Stripes. For some queries (*e.g.,* b4, b6, c2), the reduction in Stripe cardinality deviates from that of the number of Stripes, an effect caused by the varying node distributions in Stripes. The same holds regarding the Stripe size reduction, where the type of the selected Stripes also favours the observed deviation. In fact, the results are more encouraging; the average reduction of Stripe size is close to 80% of the total size of the striped dataset, while the worst case is a reduction by a factor of 45%. The normalised usage of Stripes along with the cardinality and size of the selected Stripes with respect to the striped dataset, is displayed in Figure 4.22.



(a) Stripe reduction



(b) Stripe size reduction

Figure 4.23: Normalised impact of Pruning, Rewriting and their combination on query input for the DBLP dataset

4.5.3.2 Impact of Stripe Pruning

The impact of Pruning and Rewriting on query input for the DBLP queries is shown in Figure 4.23. The results, are normalised with respect to the number of Stripes and Stripe size that are used when neither Pruning nor Rewriting are applied. We make the following observations:

Pruning The added impact of Pruning with respect to the naïve approach further re-

duces the number of Stripes by 48% on average for all DBLP queries. In particular, for queries a6, a7, b3, b5, c1 and c5-e2, the number of the selected Stripes is reduced by more than 90% on average, as seen in Figure 4.23(a); these queries contain a "//" operator, for which the naïve approach selects a large number of Stripes. Pruning, effectively discards those Stripes that are not needed for query evaluation, taking into consideration the overall query input requirements. However, the reduction in number of Stripes is not always reflected in the Strip size reduction, as can be seen in Figure 4.23(b). From the queries listed above for which a large number of Stripes is reduced, a proportional Stripe size reduction occurs for queries a6, a7, c1 and d1-e2. For the remaining queries (b3, b5, c5 and c6), Pruning does reduce the Stripe size but to a smaller extent. For the rest of the DBLP queries, Pruning has insignificant impact. Query c3 presents a special case as the impact of the number of Stripes being pruned significantly deviates from that in terms of Stripe size. This is due to the fact that the pruned Stripes have a small size compared to the total size of the striped dataset. In summary, the average reduction in Stripe size achieved by Pruning for all DBLP queries is close to 40%.

Rewriting The impact of Rewriting for the DBLP dataset with respect to the naïve approach, is also significant. As shown in Figure 4.23(a), Rewriting manages to prune a large number of Stripes for most of DBLP queries. An interesting observation is that the effect of Rewriting in reducing the number of required Stripes rather complements the effect of Pruning. For instance, for queries a1-a5, where Pruning has no effect on the number of Stripes, Rewriting achieves a sizeable reduction. The same holds, to a smaller extent though, for queries b1, b2, b4, b6, c2 and c4. On the other hand, Rewriting achieves no reduction for queries c1, c5 and c6, where Pruning seems to have a big impact, by reducing a large portion of the selected Stripes. For the remaining DBLP queries, Rewriting has the same effect as Pruning. The average reduction of the number of Stripes is close to 60% compared to the number of Stripes being selected by the naïve approach, and thus it prunes 10% more Stripes on average than Pruning. When considering the impact on the Stripe size, though, only some of the queries will actually benefit from Rewriting. This is depicted in Figure 4.23(b), where it is shown that the reduction of the number of Stripes due to Rewriting has rather minimal impact on the size of the remaining Stripes that will be accessed for

queries b1, b2, b4, c3 and c4. Nevertheless, the average Stripe size reduction for DBLP queries is at 40%, which although deviates from the equivalent reduction in terms of number of Stripes, it yields, however, a sizeable reduction in I/O cost.

Pruning and Rewriting Since the effect of Pruning and Rewriting are complementary for some of the DBLP queries, we expect additional benefits when both of them are combined. Indeed, this is shown in Figure 4.23(a), where the achieved reduction in the number of Stripes, benefits from either Pruning (*e.g.,* queries c5, c6) or Rewriting (*e.g.,* queries a1-a5). This added effect is also reflected by the average number of Stripes being reduced for all DBLP queries, which reaches the 70% of the total number of Stripes being selected by the naïve approach. The Stripe size reduction is also expected to be a combination of the effect of both Pruning and Rewriting. The combined effect in Stripe size is shown in Figure 4.23(b) and it is reduced by more than 60%, *i.e.,* 20% more than the reduction achieved from any of Pruning, Rewriting.

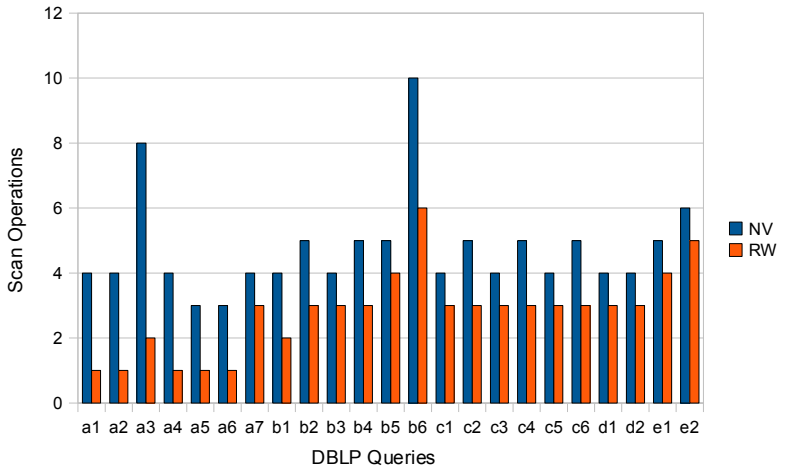
4.5.3.3 Impact of Operations Pruning

We now report the impact of Rewriting in terms of operations reduction as depicted in Figure 4.24. Similar to the other two datasets, the average reduction of Scan operations is proportional to the average reduction of number of Stripes, achieved by Rewriting. For the DBLP dataset, Scan operators are reduced by 40%, while the Structural Join operation reduction is almost 70% compared to the naïve approach. In particular, for queries a1-a5 and b1, all Structural Join operators are removed, while for all other queries at least half of them are pruned.

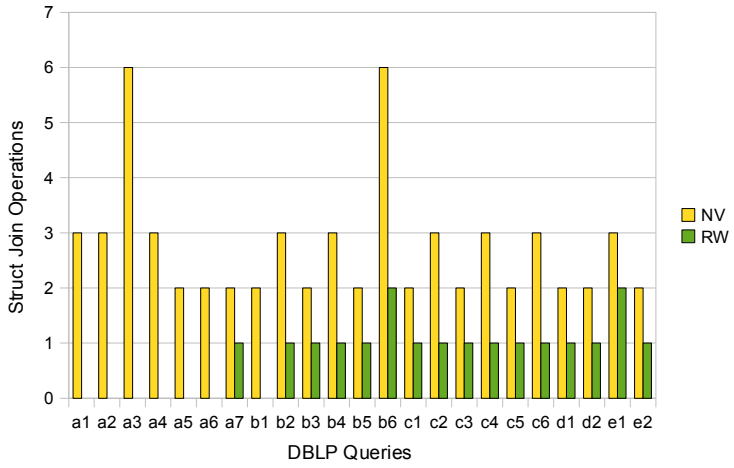
4.5.3.4 Query Performance

We now present the query evaluation performance of our native store prototype SRX for the DBLP dataset. Once again, we merely focus on the largest document produced for the DBLP dataset, (sf=10, size=1GB). The query evaluation times of all evaluation setups for the DBLP10 dataset are shown in Table 4.5. We highlight the best time of all considered setups for each of the tested queries.

We begin our analysis focusing on the impact of applying each of the optimisations PR, RW and OP in isolation. The results for each of the evaluation setups considered are presented in Figure 4.25.



(a) Scan operations reduction



(b) Structural join operations reduction

Figure 4.24: Impact of Rewriting with respect to operator reduction for the DBLP dataset

Regarding the PR evaluation setup, we observe that it performs substantially better compared to the NV setup for ten of the DBLP queries, namely a6, a7, b3, b5, c1, c5, c6, d1, e1 and e2. For the rest of the DBLP queries, though, the evaluation times for the PR setup are at the same level as the ones of the NV. Nevertheless, the average speedup achieved by the PR evaluation setup is almost 40%, as much as the reduction accomplished regarding the Stripe size, which effectively justifies the evaluation speedup. An interesting result is that of query d2: The evaluation time for the specific query is almost the same as the one achieved by the NV setup, although the Stripe size reduction performed by Pruning is significant. The reason for that, is that the total number of Stripe pages of the Stripes selected for query evaluation, is relatively small,

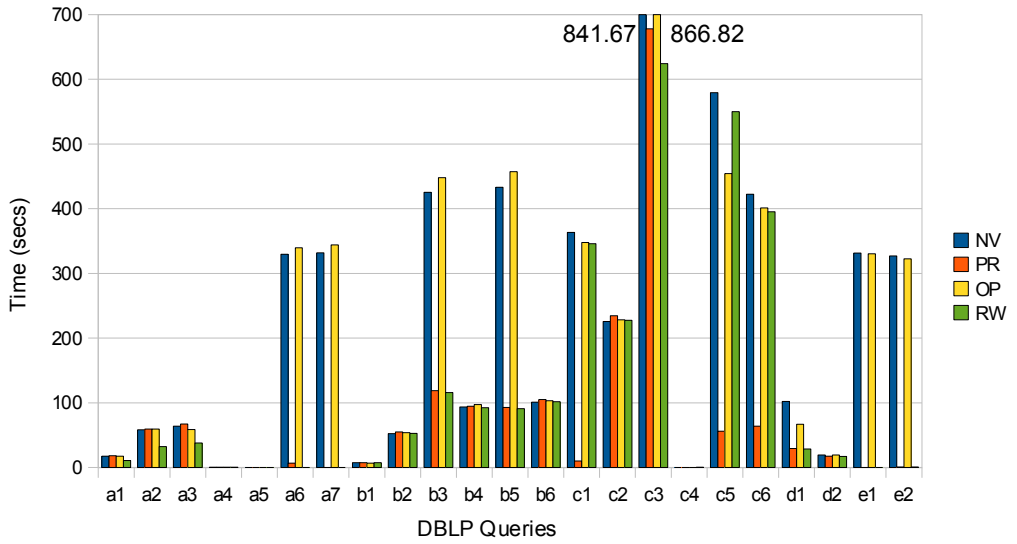


Figure 4.25: SRX query evaluation results for each optimisation in isolation

and as a result the reduction of the number of Stripes is not reflected in the already fast evaluation of query d2.

The OP evaluation setup, does not have a noticeable impact on query evaluation. The evaluation times are almost the same as the ones of the NV setup for most of the tested queries, while the average speedup is in the order of 5%. The queries for which the applied Stripe-aware Optimisation has noticeable impact on running time, are queries c1, c5, c6 and d1, with c6 and d1 having the bigger improvements at evaluation times: 20% and 35% respectively. To gain an insight in the benefits of the OP evaluation setup, we have measured the number of Stripe nodes being retrieved for both the NV and OP setups; the normalised results of OP, with respect to the NV setup, are displayed in Figure 4.26. Note that the reduction of the number of nodes being accessed is not an absolute I/O metric as what is mostly important, is the number of retrieved pages from disk. However, the number of reduced nodes may still provide a hint of the benefits of Stripe-aware algorithms as a reduction of the total number of retrieved nodes by a large factor also implies a reduction on accessed Stripe pages. This is the case for queries c1, c5 and c6, where a reduction of more than 60% of the total Stripe nodes being accessed occurs, which is reflected in their evaluation time. For the rest of the queries, a small or average reduction in retrieved Stripe nodes, has minimal or no impact at all on the evaluation time, as it does not necessarily reduce the number of Stripe pages, being accessed. An exception to that is query d1 whose running time is improved by a large factor despite the rather average number of nodes

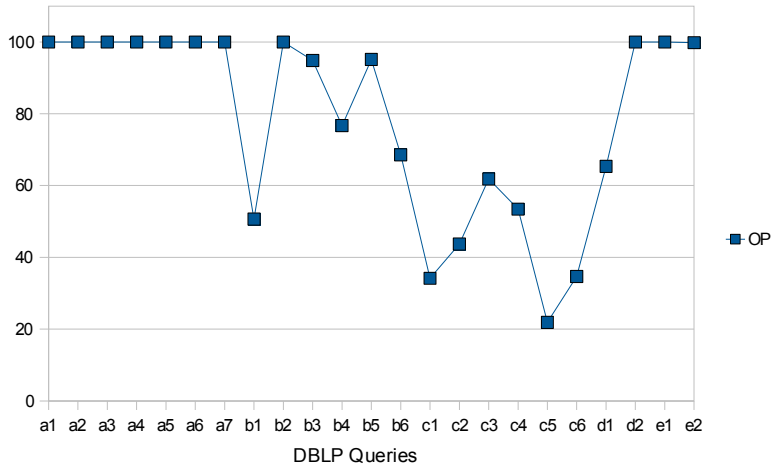


Figure 4.26: Normalised access to Stripe nodes when Stripe-aware Optimisation is applied for the DBLP Dataset

being reduced. For the evaluation of query d1, a Stripe-aware following-axis Structural Join operator is employed, which effectively reduces a number of nodes being continuously stored in Stripe pages. Therefore, the reduction of number of Stripe pages is proportional to the reduction on Stripe nodes and thus impacts evaluation time. On the other hand, for some DBLP queries, namely a6, a7, b3, b4, b5, c2 and c3, the OP setup performs worse than NV by a small factor. This happens due to the fact that certain Stripe-aware algorithms are computationally more expensive than their Stripe-unaware counterparts, in their attempt to avoid access to Stripe nodes that are not needed. Thus, when the query I/O cost is not reduced, the total evaluation time is burdened with the extra computational cost of (some of) the Stripe-aware algorithms.

The RW evaluation setup has a significant impact on query evaluation times in comparison to the NV setup. The accomplished speedup on evaluation times is 35% on average for all DBLP queries. In particular, seven queries (a6, a7, b3, b5, d1, e1 and e2) are improved by more than 75% while six others (a1-a5 and c3) from 20% to 40%, compared to the NV evaluation setup. The performance benefits of the RW evaluation setup are caused primarily by the Stripe size reduction, and secondarily by the reduction of physical operators, as described in Section 4.5.3.2 and Section 4.5.3.3 respectively.

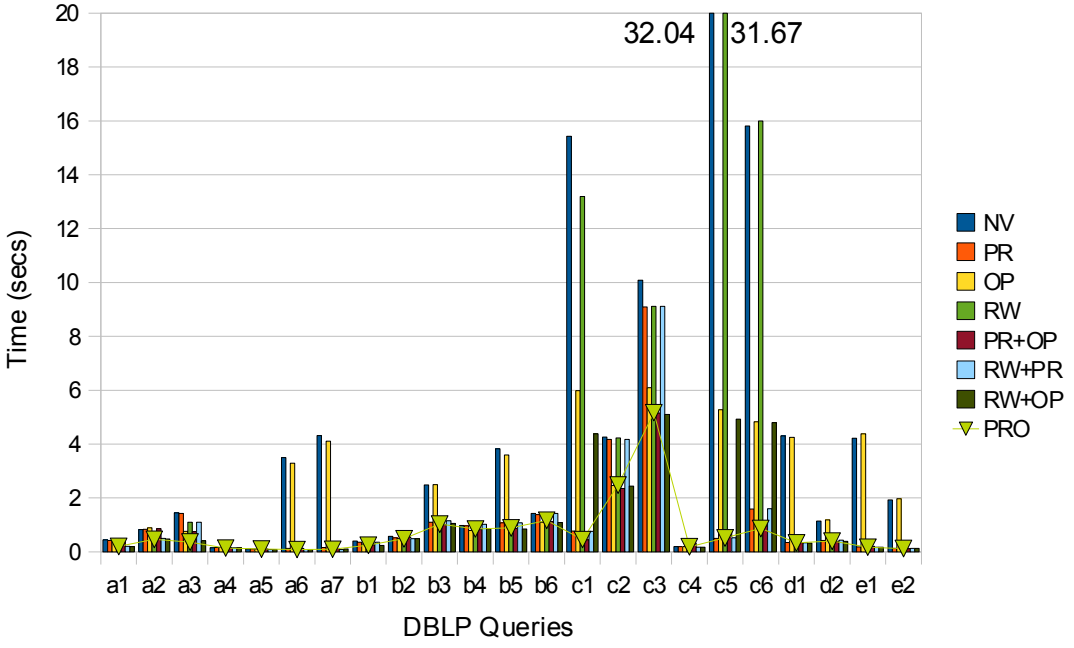
We now turn our attention to the combined evaluation setups. For the PR+OP and RW+OP evaluation setups, the evaluation times for most DBLP queries are primarily dominated by the effect of applying Pruning or Rewriting optimisations, respectively.

The average speedup achieved by the PR+OP setup with respect to the NV setup is a bit more than 40%; very close to the improvement achieved by the PR setup alone. The added effect of Stripe-aware algorithms merely improves query evaluation times by 1,5% on average. The combined effect of the PR+OP setup is evident only in query c3, for which the evaluation time is improved by a further 8% compared to the PR setup. For the rest of the DBLP queries, the added effect of Stripe-aware Optimisation is rather minimal. Similar observations hold for the RW+OP evaluation setup. Rewriting is the dominant factor on evaluation times but Stripe-aware Optimisation has now a greater impact when combined with Rewriting than the PR+OP setup, when they are combined with Pruning. The RW+OP evaluation setup achieves an average speedup of 40% compared to the NV setup and therefore Stripe-aware Optimisation further reduces evaluation time by 5% compared to the RW evaluation setup. As for the RW+PR evaluation setup, there does not seem to exist a dominant optimisation, since Pruning and Rewriting both overlap and complement each other in terms of their Stripe reduction effect. We thus expect this to be reflected in the evaluation times of the DBLP queries. Indeed, we observe that for most of the queries, the evaluation time of the RW+PR evaluation setup is either improved due to the Pruning optimisation effect (*e.g.,* queries c1, c5, c6) or due to the Rewriting optimisation effect (*e.g.,* a1, a2, a3). The RW+PR evaluation setup achieves a combined average speedup of 50% improving both the PR and RW evaluation times. Finally, for the PRO evaluation setup which combines all optimisations, query evaluation times are affected by the optimisation that is the most dominant for each query. The evaluation times of the PRO setup, deviate from the best evaluation result by 4.4% on average. The results for all tested optimisation setups running on the largest DBLP dataset considered (DBLP10) are shown in Figure 4.28.

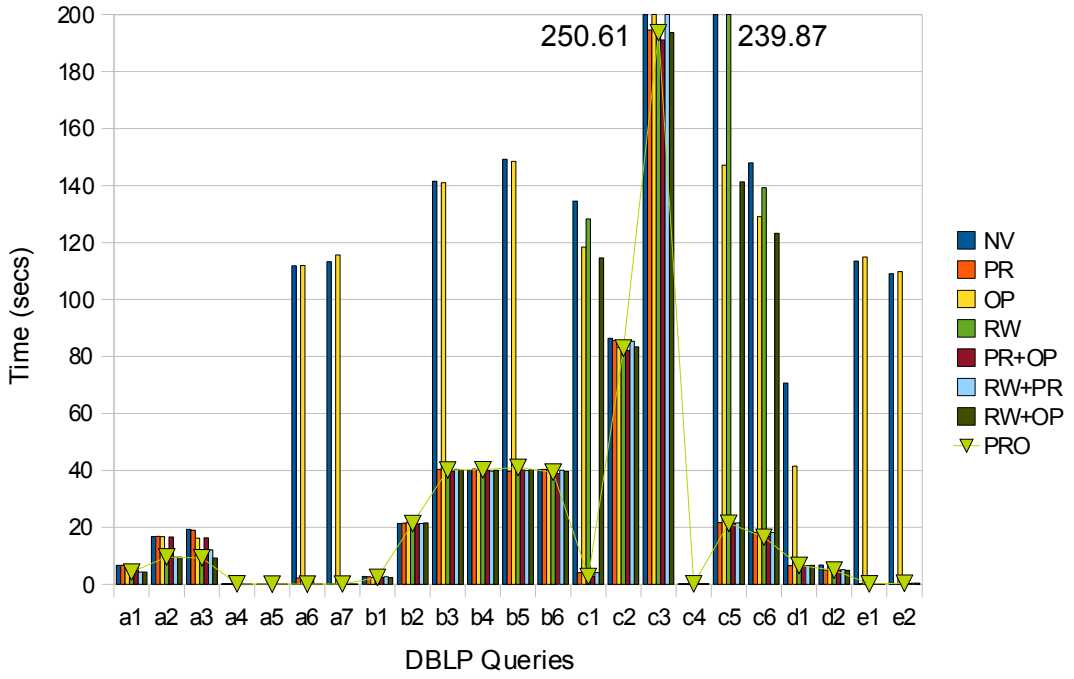
4.5.4 Comparison with MDB

We now compare our evaluation engine prototype over the explicit storage scheme, SRX, with the state-of-the-art in XML query evaluation MonetDB/XQuery, referred to as MDB. We briefly review both systems, identifying their main advantages and weaknesses.

The design of our native striped XML store, SRX, is based on Striping. SRX's main advantage is the large degree of fragmentation that Striping imposes; this effectively



(a) DBLP1



(b) DBLP5

Figure 4.27: SRX query evaluation for the DBLP1 and DBLP5 datasets

provide the ability to minimise the query input and therefore reduce I/O by a large factor. However, in rare cases this large degree of fragmentation may have a negative

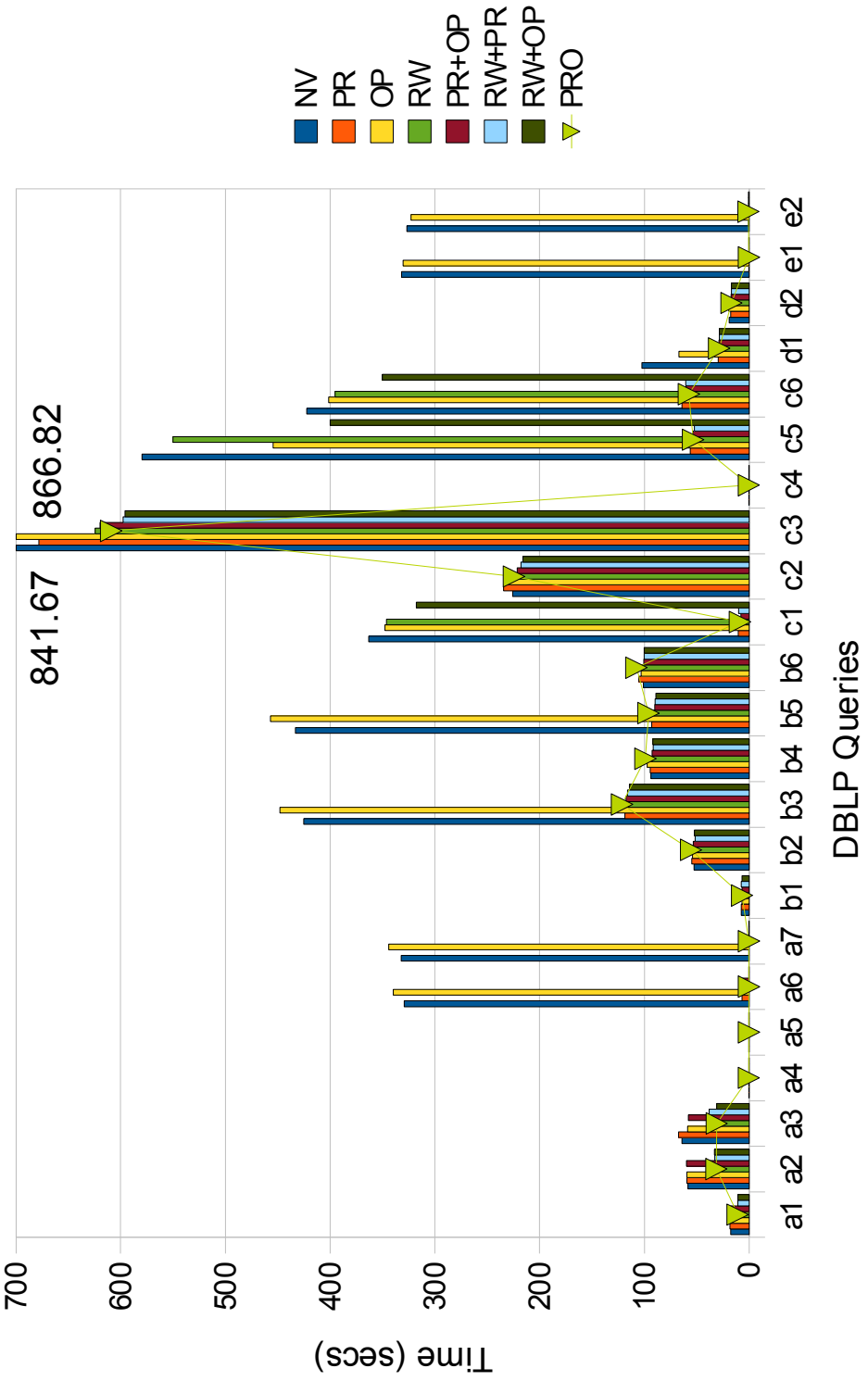


Figure 4.28: SRX query evaluation for the DBLP10 dataset

Query	DBLP10 (1 GB)							
	NV	PR	OP	RW	PR+OP	RW+PR	RW+OP	PRO
a1	17.67	18.14	17.56	10.87	17.62	10.82	10.81	**10.79**
a2	58.29	59.39	59.33	32.24	59.8	32.7	33.01	**31.24**
a3	63.95	67.26	58.9	37.99	57.82	37.95	31	**30.94**
a4	0.29	0.3	0.3	**0.21**	0.3	**0.21**	**0.21**	0.22
a5	0.16	0.16	**0.12**	0.13	0.14	**0.12**	0.14	0.13
a6	329.42	6.64	339.64	0.14	6.48	**0.13**	0.15	**0.13**
a7	332	0.16	343.94	0.13	0.15	0.15	**0.12**	0.14
b1	7.43	7.49	6.94	7.52	6.83	7.32	6.82	**6.77**
b2	52.55	54.84	53.97	52.63	53.39	**51.63**	52.2	55.44
b3	425.27	118.68	448.04	115.77	117.62	116.07	**114.26**	121.38
b4	93.63	94.5	97.28	92.68	92.51	**91.65**	91.91	98.92
b5	433.06	92.81	457.11	91.02	90.16	89.68	**89.06**	96.02
b6	101.06	105.16	103.08	101.91	100.63	**100.09**	100.15	107.5
c1	363.29	10.23	347.82	346.1	**7.8**	9.92	317.76	8.89
c2	225.65	234.43	228.08	227.64	221.29	217.82	**216.03**	224.44
c3	841.67	678.11	866.82	624.43	619.59	597.64	**595.94**	609.09
c4	**0.21**	**0.21**	**0.21**	0.23	0.22	0.23	**0.21**	0.24
c5	579.5	56.25	454.27	550.16	53.79	**52.28**	400.05	53.41
c6	422.43	64.02	401.34	395.3	60.33	60.42	350.36	**57.54**
d1	102.21	29.56	66.82	28.67	28.74	28.43	**28.28**	28.49
d2	19.16	17.44	19.23	17.23	17.19	16.86	**16.74**	16.79
e1	331.67	**0.2**	330.44	**0.2**	**0.2**	0.21	0.22	**0.2**
e2	326.91	0.79	322.73	0.73	0.67	0.71	**0.66**	**0.66**

Table 4.5: SRX performance for DBLP queries

impact on query performance; extra Stripe merging operations are in need. The most evident SRX weakness though, is the lack of content indexing. If a value-based predicate exists, the whole Attribute/Value Stripe is scanned, regardless of the predicate's selectivity.

MonetDB/XQuery, MDB, is an open source XML database system that fully supports the XQuery language [92] and thus XPath 2.0 [14]. *PathFinder* was first introduced as a tree-aware, database index proposal for accelerating XPath location steps [47]. After a short time, a new, location step evaluation algorithm was proposed as a database kernel extension, the *Staircase Join* [49], that further increases the level of tree awareness and improves the evaluation of XPath location steps. Although PathFinder was initially designed for RDBMSs with conventional index structure support such as B-trees

or R-trees, it also fitted on top of the main memory database kernel, *MonetDB* [16], providing encouraging results [50]. Later, PathFinder evolved to a full-fledged XQuery compiler [48], translating XQuery expressions to relational algebra. MDB has evolved over the years to become the state-of-the-art XML database system. MDB combines the advantages of both PathFinder and MonetDB. In addition to the proposed range-based encoding of XML documents, the optimised, tree-aware, evaluation algorithms for efficient evaluation of XPath expressions and the translation of XQuery expressions to purely relational algebra (PathFinder), MonetDB has a big share in MDB's success. MonetDB is designed to implement a binary relational model, *i.e.,* data is stored in binary tables that emulate a vertically (column) partitioned relational table. This enables MonetDB to minimise I/O by retrieving only the decomposed binary tables that are needed. The drawback of the vertically-partitioned design is the need for extra join operations to re-assemble the fragmented tuples; MonetDB, however, exploits the modern CPU architecture and memory cache hierarchies to provide tailored operators for binary table joins. This additional computational cost is most of the time smaller than the I/O cost of accessing non-fragmented data that usually contain more information than what was requested. For MDB, the encoding table of the XML documents is shredded based on MonetDB's binary relational model. Another advantage of MonetDB is that being a main memory database, it uses the available main memory to materialise intermediate results. This may have a huge impact on overall evaluation time, since sorting, merging and duplicate elimination operations can now be efficiently performed. This also extends to MDB; as long as the produced intermediate results (XML sequences) fit in main memory, no extra I/O operations are performed. In addition, this enables MDB query operators and algorithms to shift away from the iterator interface paradigm to a set-at-a-time processing that operates on a sequence of context nodes in a single step, producing a new XML node sequence. Such algorithms are more efficient compared to the iterator-based ones because (a) they make better use of modern CPU architecture and memory cache [49] and (b) they do not require extra synchronisation cost.

4.5.4.1 Xmark

We present the evaluation times of both systems for the largest Xmark dataset (sf=100, size=11GB), as shown in Figure 4.29. According to the results, neither system performs better than the other for all Xmark queries. For queries a2-a4, b1-b3 and d1-d2, that is 8 out of 20 Xmark queries or 40% of the query testbed, SRX outperforms MDB.

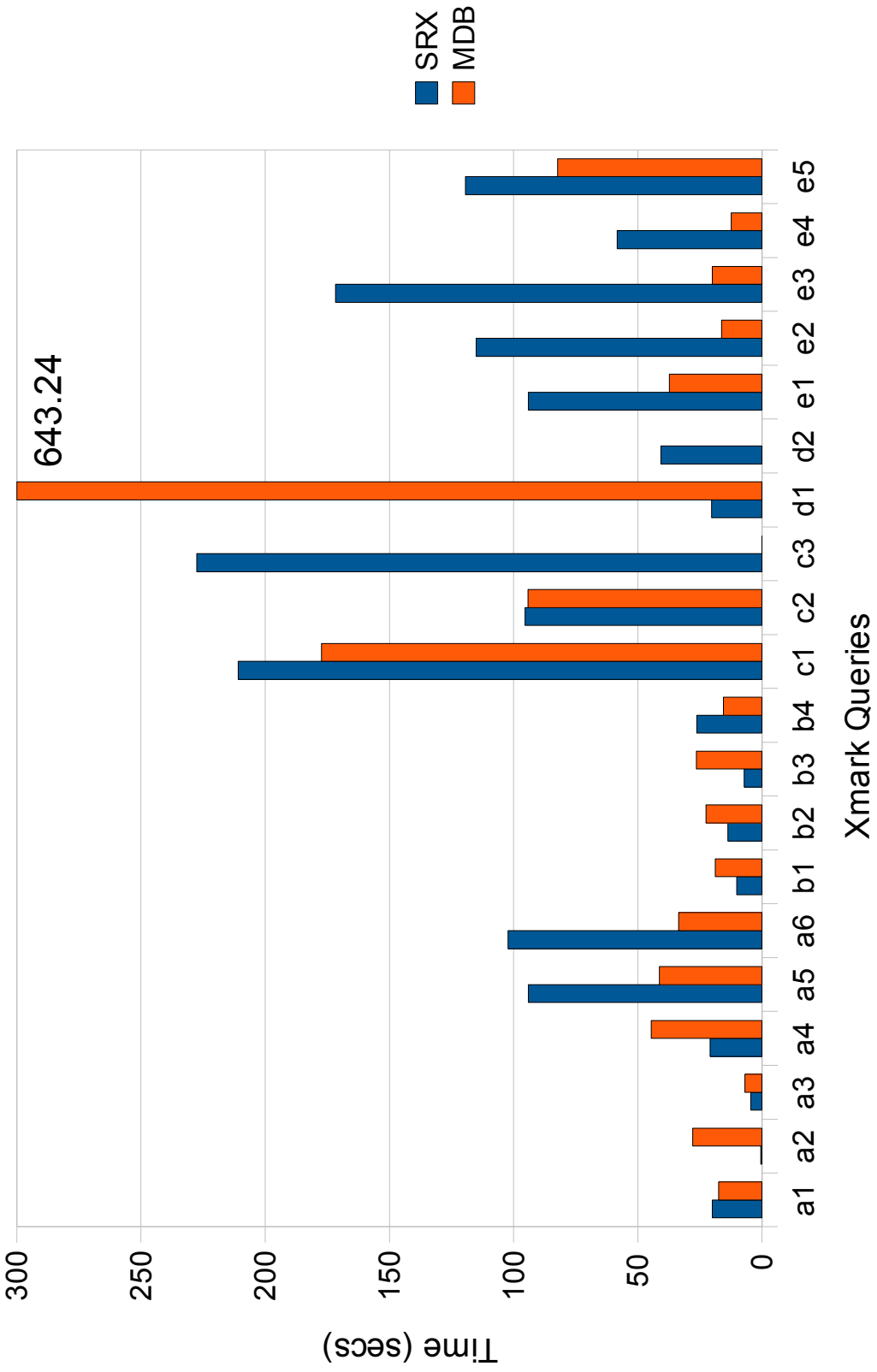


Figure 4.29: SRX and MDB comparison for the Xmark100 dataset

On the other hand, MDB outperforms SRX for queries a5-a6, b4, c1, c3 and e1-e5, 50% of the Xmark query testbed. Finally, for queries a1, c2 both systems have comparable performance.

Queries a1-a6 of the Xmark testbed are queries that focus on parent-child relationships of XML nodes. For queries a2 and a4, SRX performs better than MDB since long sequences of parent-child relationships are effectively reduced to a minimal set of Stripe scans and structural join operations due to our striped-based decomposition method. On the other hand, MDB outperforms SRX for queries a5, a6; query a5 involves a very selective attribute predicate (which returns exactly one result) and SRX is doomed to underperform since it lacks value indexing. Query a6 also contains an attribute-based predicate but the predicate merely involves their structural information and not the actual attribute values. Since such information is coupled with the Attribute Stripes, SRX will also access the attribute values, although such an operation is needless; MDB, on the other hand, due to the vertical fragmentation that the underlying storage model enforces, is able to merely access the attribute structural information without the actual attribute values and thus substantially reduce the query input size.

Queries b1-b4, on the other hand, focus on XML node ancestor-descendant relationships. SRX performs better than MDB for queries b1-b3, due to the application of query equivalence rules over its striped model (Rewriting) that effectively reduce the query input size to the minimum required. However, SRX performs significantly worse than MDB at query b4. Query b4, involves a predicate expression that due to a descendant axis location step expression, uses a Merge Scan access method operator over nine Stripes. SRX's inefficiency lies in the extra merge operation over nine input Stripes for providing the candidate nodes for the location step expression. In addition, for query b4, SRX has no benefit on any of its proposed optimisations (Pruning, Rewriting, Stripe-aware Optimisation).

Queries c1-c3 of the Xmark testbed focus on the evaluation of XML nodes sibling relationships. However, queries c1 and c3, also involve the evaluation of one and two very selective value-based predicates, respectively. MDB's optimiser correctly chooses to use value indexes to access attribute values and for that reason it outperforms SRX that applies selections as post-filters on top of Attribute Scans. For query c2, however, where there is no value-based predicate, both SRX and MDB perform the same.

We continue with Xmark queries d1-d2, that focus on XML node preceding-following relationships. SRX is a clear winner in this query category; the reason that MDB underperforms is the large intermediate results that are produced from the evaluation of

the following axis location steps. This becomes clearer for query d2, where MDB was unable to return a result due to its main memory limitation. SRX, on the other hand, that implements iterator-based operators and produces non-blocking evaluation plans, was able to perform significantly better than MDB.

Finally, Xmark queries e1-e5 focus on XPath predicate expressions that contain boolean operators. In this query category, MDB outperforms SRX by a large factor. Although for these queries, we do achieve (due to Striping) a significant I/O reduction compared to the whole striped representation of the document, Striping (*i.e.,* path-based decomposition) does not provide further savings from what a tag-based decomposition would provide. Thus, for queries e1-e5, SRX has no real benefits from its most distinctive feature that affect query performance the most. MDB, on the other hand, using selections on tag attributes over the fragmented document encoding can achieve the same affect as SRX in terms on input reduction. It is also clear that MDB evaluates boolean predicates (using set operators on memory resident intermediate results) in a more efficient way compared to the iterative, context-node driven predicate evaluation of SRX.

4.5.4.2 Mbench

We proceed with the evaluation times of SRX and MDB for the largest Mbench dataset (sf=10, size=5GB). As already described, Mbench datasets are mainly characterised by their deeply nested and highly recursive structure. This makes Mbench datasets, good candidates for testing complex structural relationships among XML elements. To that end, most Mbench queries mainly involve (a) step axis expressions that actually test such structural relationships and (b) value-based predicates for controlling the selectivity of the structural relationships. When very selective value-based predicates are used, SRX has a disadvantage compared to MDB since it does not employ any kind of value indexing. However, SRX employs content-aware projection Section 2.3.2.3 as part of Input Minimisation; each of the candidate Attribute/Value Stripes are tested whether they contain values that may produce a match based on the value-based predicate. If a Stripe is guaranteed not to contain the requested value(s), then it is discarded (pruned). In addition, when an Attribute/Value Stripe is pruned, it may trigger further pruning of Path Stripes according to the Stripe Pruning process, as described in Section 2.3.3. This has a huge impact on the Mbench dataset as for most tested queries, a significant amount of Stripes is pruned.

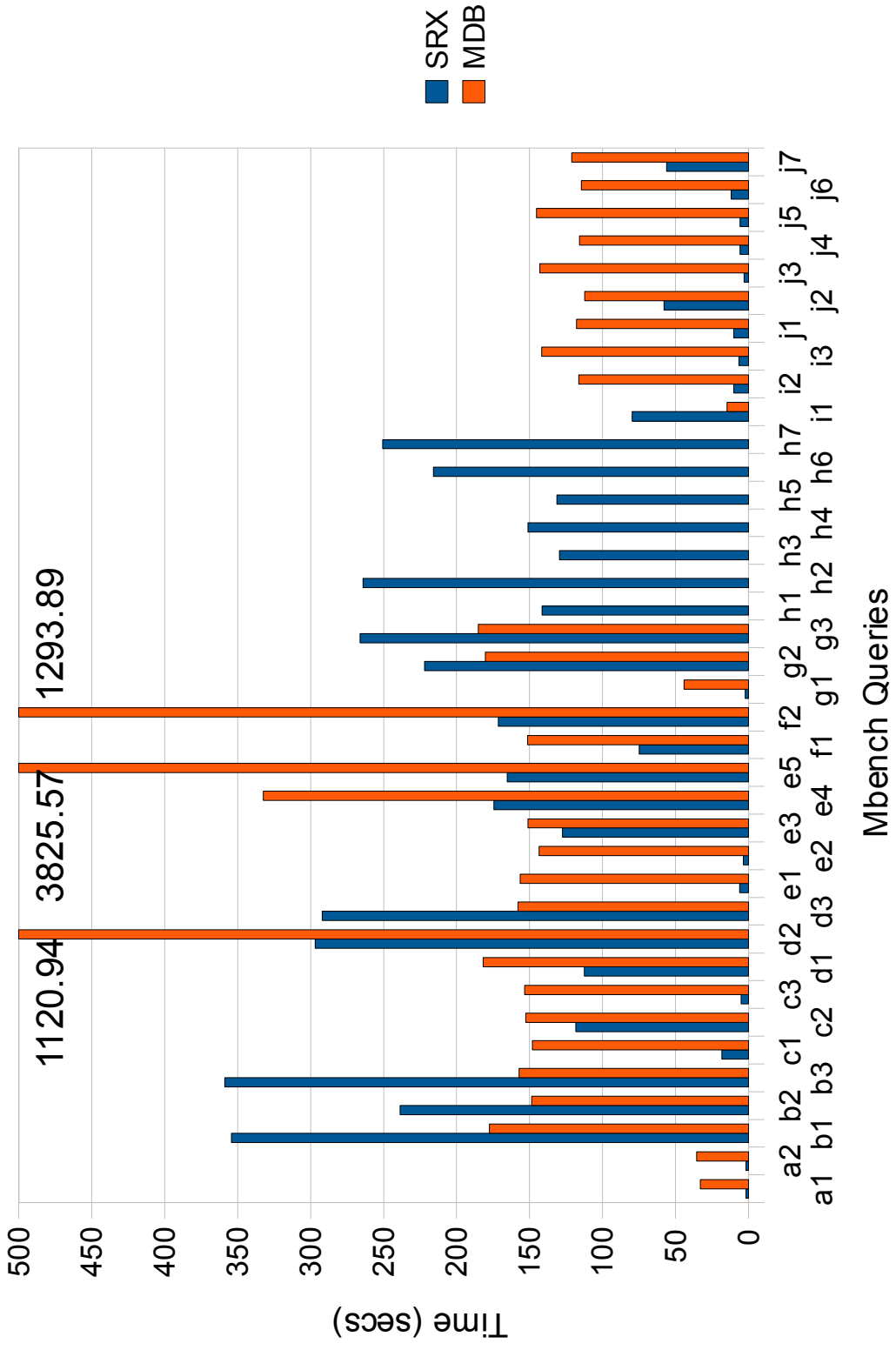


Figure 4.30: SRX and MDB comparison for the Mbench10 dataset

The evaluation results for both systems are shown in Figure 4.30. For the majority of Mbench queries, *i.e.,* 31 out of 38 Mbench queries, (nearly 82% of the query testbed), SRX outperforms MDB. MDB performs better than SRX for the remaining 7 Mbench queries: b1-b3, d3, g2-g3 and i1.

Queries a1-a2, produce the same result: elements that occur occasionally in the Mbench document structure; such selection is based on element tag name. For such queries, SRX performs better than MDB, since due to Striping and the equivalence rules, it selects only the required Stripes and thus minimises I/O.

However, for queries b1-b3, where the selective predicate is a value-based predicate with varying selectivity, MDB outperforms SRX; As already explained, SRX lacks content indexing which results in full Stripe Scans. Observe that the SRX evaluation times for b1 and b3 queries are very close, since they involve the same set of Stripes and the evaluation process is the same regardless of the predicate selectivity. For query b2, SRX performs better compared to queries b1 and b2 due to the smaller size of the selected Attribute Stripe.

Queries c1-c3 and d1-d3 test parent-child and ancestor-descendant XML node relationships, respectively, with varying selectivity of parent/ancestor and children/descendant nodes. For queries c1-c3 and d1, SRX outperforms MDB due to the large impact of Pruning on the selected query input. Note that for query c2, where the performance of SRX is close to that of MDB, the Pruning impact is less evident compared to queries c1, c3. For queries d2 and d3, SRX has no significant benefit from Stripe Pruning and thus the evaluation times for both queries are similar since full Attribute Scans are employed regardless of the value selectivities. However, this does not apply to MDB, where the varying selectivities directly affect evaluation times. The less selective ancestor node predicate in query d2 seems to be the reason for that, since a large intermediate result sequence is produced; for query d2, MDB performs almost 4 times worse than SRX. On the contrary, the highly selective ancestor node predicate of query d3, boosts MDB's performance, which outperforms SRX by almost a factor of 2.

Queries e1-e5 are complex twig pattern matching queries with mixed parent-child and ancestor-descendant XML node relationships of varying selectivities. Similarly, queries f1-f2 are complex twig pattern matching queries with one long branch (chain) of parent-child only or ancestor-descendant only relationships of varying selectivities. For all queries e1-f2 and regardless selectivities, SRX outperforms MDB due to the impact of Pruning on input size that directly reflects SRX's query evaluation times. The Pruning impact is even more evident for queries e1-e2, that merely involve parent-

child only node relations; These are exploited by the Pruning process that effectively reduces SRX input size by more than 95%. On the other hand, for queries e3-e5 that also involve ancestor-descendant node relationships on the highly recursive structure of the Mbench dataset, Pruning has smaller impact on input size reduction, although still significant as reflected on query evaluation times. Similarly, Pruning has a larger impact on query f1 that involves parent-child only relationships, compared to f2 that these are replaced by ancestor-descendant relationships. Regardless of the degree of Pruning efficiency, however, the I/O input size reduction, is important enough so that SRX performs better than MDB, despite the lack of value-based indexing.

Queries g1-h7 are adaptations of some of the above queries (a1-f2), only that a parent-child or ancestor-descendant relationship is replaced by a preceding-following XML node relationship. Query g1 resembles queries a1, a2 in that the node selection is based on their tag names and thus Striping and equivalence rules have a significant impact which is reflected on query evaluation time. For queries g2-g3 though, MDB outperforms SRX; the evaluation of the preceding-following relationship in queries g2-g3, is dominated by the selection of candidate nodes which are in turn dominated mostly by the value-based predicates. Thus, SRX underperforms due to its poor performance on value-based selections. For queries h1-h7, the preceding-following relationship occurs within a filter predicate. MDB was unable to process any of those queries due to the large intermediate results, produced from the following axis location step. Finally, queries i1-j7 are created in the same manner as queries g1-h7, except that the preceding-following relationships are now replaced by a sibling XML node relationship. Again, query i1 resembles queries a1, a2 and g1 in that the node selection is based on their tag names. In this case, however, due to the sibling node relationship, the benefits of Striping, Pruning and Rewriting is minimal. For the rest of the queries that contain sibling relationships though, the large impact of Pruning is reflected on SRX query evaluation performance, outperforming MDB.

4.5.4.3 DBLP

We conclude with the evaluation results of SRX and MDB for the DBLP dataset. As already described, the DBLP datasets are fragments of a bibliography database of Computer Science journal and conference proceedings. The dataset is structured in such a way so that there exists a sequence of entries, directly under a top level element that describes the type of a bibliographical entry. As a result, the DBLP structure is a “wide”

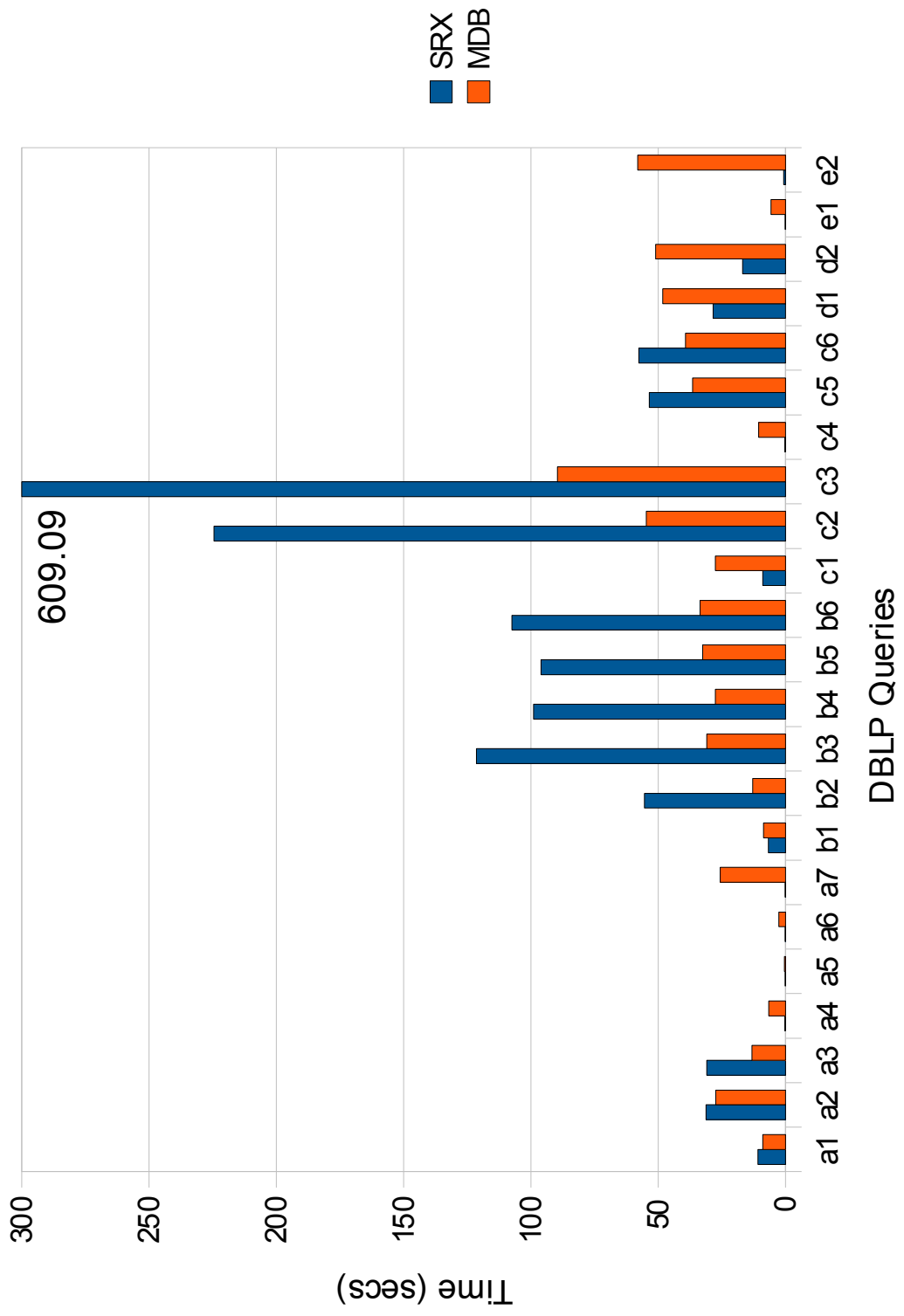


Figure 4.31: SRX and MDB comparison for the DBLP10 dataset

tree having as many children nodes at level 1, as the number of bibliography entries. To that end, most DBLP queries mainly involve node selections based on (a) tag names and (b) content-based predicates with the latter being the dominant expression for most queries of the DBLP testbed.

The query evaluation results for the largest DBLP dataset (sf=10, size=1GB), are shown in Figure 4.31. Queries a1-a7 of the DBLP testbed are queries that focus on tag-based selection of XML nodes and mainly involve parent-child (a1-a5) and ancestor-descendant (a6-a7) node relationships. SRX performs better than MDB for queries a4-a7, where Striping combined with Pruning and Rewriting, enables SRX to minimise query input to the maximum extent. On the other hand, MDB outperforms SRX for query a3 as it employs superior set operations over binary relations. For queries a1-a2, both systems have comparable results, with MDB performing better by a small factor though.

Queries b1-b6, on the other hand, focus on value-based selection of XML nodes; they also involve mixed parent-child and ancestor-descendant node relationships. MDB is the clear winner for queries b2-b6, as they contain very selective value-based predicates for which SRX underperforms. Query b1 is the only exception in this query category due to the less selective value-based predicate. Queries c1-c6 focus on wild-card operators but similarly to b1-b6 queries, they also involve value-based selections (except query c1). For query c1, where no value-based predicate operation occurs, SRX performs better than MDB. The same holds for query c4, where despite the value-based predicate, due to Pruning, SRX restricts the overall I/O cost to a very low value. However, for queries c2, c3 where Pruning and Rewriting have minimal effect on input size reduction, MDB outperforms SRX by a large factor. MDB also performs better than SRX for queries c5-c6, due to the selective value-based predicates; however, SRX, due to Pruning, manages to restrict query input size so its results for queries c5-c6 are relatively close to those of MDB.

Finally, queries d1-d2 and e1-e2 involve preceding-following and sibling node relationships respectively, in conjunction with value-based predicates. For such queries, SRX outperforms MDB by reducing the input size by a large factor, due to Striping and Pruning and despite the presence of value-based predicates.

Chapter 5

Tree-Sharing Compression Storage Scheme

In the previous chapter, we described a natural storage scheme over the general, striped decomposition model. We now turn our attention to exploiting structural regularities that are present in XML documents, to compress their structural part in a more compact representation. Our ultimate goal is to minimise Path Stripe storage cost and, thus, the I/O cost during query evaluation.

For this purpose, we propose the *tree-sharing* compression storage scheme, which is based on the compression technique of sharing common subtrees. This has been proposed for creating main memory structural summaries of XML document trees [20, 19] and was briefly presented in Section 1.1.3.1. The outcome of serialising the compressed, structural summary which is based on subtree sharing, to the striped decomposition model (tree-sharing scheme) corresponds to serialising the document tree structure to the striped decomposition model (explicit scheme).

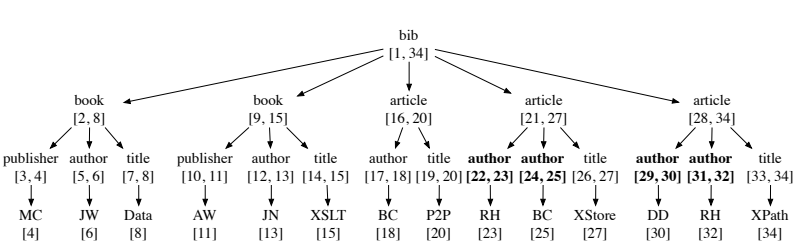
The rest of the chapter is organised as follows: In Section 5.1, we present the tree-sharing compression technique, while in Section 5.2, we discuss its main storage characteristics. In Section 5.3, we present the loading process, while in Section 5.4, we describe how nodes are reconstructed from the compressed Path Stripe nodes. Finally, in Section 5.6, we present the experimental results of the proposed storage scheme, including the compression effectiveness and its impact on query evaluation performance.

5.1 Structural Compression

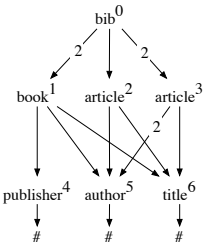
In this section, we introduce a tree-sharing storage scheme that exploits structural regularities that are present in an XML document, in order to serialise its structural part in a more concise way. Let us for example consider the XML tree depicted in Figure 5.1(a). One can identify that its structure is highly repetitive, *e.g.,* the first two `book` nodes have identical structure and also occur consecutively. The same holds for the last two `article` nodes and the two `author` nodes nested in each of them. This is not a rare example, especially if we consider the amount of XML documents being published from relational databases. We aim to exploit such properties, whenever applicable, to minimise the storage cost of Path Stripes and, as we will see, the I/O cost during query evaluation.

We identify common subtrees based on the notion of (forward) *bisimilarity*, as defined in [20]. According to this, common, consecutive subtrees in the tree representation can be identified in a bottom-up fashion, and then collapsed into a single instance. This instance is then annotated with multiplicity information *i.e.,* the number of occurrences collapsed into the instance. To reflect this in the striped representation, we create shared Path Stripe nodes N that correspond to multiple tree nodes, according to subtree sharing. To accommodate node sharing, we need to store extra information. Each shared node is assigned: (a) the *start* position of the first tree node being shared, (b) the *end* position of the last tree node being shared, (c) the *par* value of the first tree node being shared, (d) a *mult* value, the multiplicity of shared node for that interval and (e) *trace*, sharing information that allows us to reconstruct the original structure. The *trace* of a *shared* node N is a sequence of $(m_i : e_i)$ pairs (separated by '/') one for *each shared tree node* along the path p from the root of the document; m_i is the node multiplicity of the shared node, while e_i is its extent, *i.e.,* the difference between its *end* and *start* positions. Note that there is a distinction between *tree node multiplicity* and *shared node multiplicity*. The shared node multiplicity accounts for the number of tree nodes being shared, and is equal to the product of all node multiplicities on path p from the root to the node of the subtree being shared.

Example 5.1: When the subtrees rooted at the first two `book` nodes are shared into a single instance, the node multiplicity of the `book` nodes is 2 and so is the corresponding shared node's multiplicity. For the shared `book` node's children though, *e.g.,* `title` nodes, we have that their node multiplicity is equal to 1, as each of the subtrees contain



(a) An example XML document



(b) The skeleton for document Figure 5.1(a)

/bib/book

[s,e]	par
[2,8]	1
[9,15]	1

/bib/book/author

[s,e]	par
[5,6]	2
[12,13]	9

/bib/book/title

[s,e]	par
[7,8]	2
[14,15]	9

/bib/article

[s,e]	par
[16,20]	1
[21,27]	1
[28,34]	1

/bib/article/title

[s,e]	par
[19,20]	16
[26,27]	21
[33,34]	28

/bib/article/author

[s,e]	par
[17,18]	16
[22,23]	21
[24,25]	21
[29,30]	28
[31,32]	28

(c) A subset of explicit Path Stripes for the document of Figure 5.1(a)

/bib/book

[s,e]	par	mult	trace
[2,15]	1	2	{2:6}

/bib/book/author

[s,e]	par	mult	trace
[5,13]	2	2	{2:6/1:1}

/bib/book/title

[s,e]	par	mult	trace
7,15]	2	2	{2:6/1:1}

/bib/article

[s,e]	par	mult	trace
[16,20]	1	1	{}
[21,34]	1	2	{2:6}

/bib/article/author

[s,e]	par	mult	trace
[17,18]	16	1	{}
[22,32]	**21**	**4**	**{2:6/2:1}**

/bib/article/title

[s,e]	par	mult	trace
[19,20]	16	1	{}
[26,34]	21	2	{2:6/1.1}

(d) A subset of compressed Path Stripes for the document of Figure 5.1(a)

Figure 5.1: Tree-sharing compression storage scheme for the striped model

a single `title` node, while their shared node's multiplicity value is again 2 as there are two tree nodes being shared overall. □

Finally, note that all nodes being shared have the same extent value e. The latter is the size of the subtree rooted at each node. More insight concerning the construction of trace for each shared entry will be discussed later at Section 5.4. Trace information is used when we need to "split" a shared node and reconstruct its constituents.

Example 5.2: A fragment of the Path Stripes built for the document of Figure 5.1(a) is shown in Figure 5.1(d). Consider the two subtrees rooted at the first two `book` nodes of the document tree. Due to bisimilarity, these subtrees can be collapsed into a single instance. The same applies to all Path Stripe nodes that correspond to the nodes of the subtrees being shared. Thus, the first two Stripe nodes of the `/bib/book` Path Stripe of Figure 5.1(c) will be shared in the first Stripe node of the corresponding Path Stripe of Figure 5.1(d); similarly, for the `/bib/book/publisher`, `/bib/book/author`, and `/bib/book/title` Path Stripes. Observe that all nestings of the original document are faithfully maintained in the new representation of Path

Stripes. We can still test structural relationships between any Stripe nodes by employing exactly the same tests according to the labelling scheme. For instance, the first Stripe node in the `/bib/book/publisher` Path Stripe is only nested in the first Stripe node of the `/bib/book` Path Stripe. Next, observe the second Stripe node in the `/bib/article/author` Path Stripe, shown in bold in Figure 5.1(d). The node's *mult* value is 4, denoting that it consists of four Stripe nodes in the non-shared version of Path Stripes (explicit scheme), or else four tree nodes (also shown in bold). Its *trace* value: $\{2{:}6/2{:}1\}$ contains two $(m : e)$ pairs, denoting two `/bib/article` shared ancestor nodes, (each with an extent of 6) that each of them contains two `/bib/article/author` nodes (each with an extent of 1). It is easy to see that this node's *trace* is derived from the second node of the `/bib/article` Stripe concatenated with its $(m : e)$ pair. □

5.2 Stripe Storage

Similar to the explicit storage scheme, all (Path, Attribute and Value) Stripes are stored in separate B$^+$-trees with the *start* value acting as the B$^+$-tree key. Thus, all Stripe nodes are stored in ascending *start* value order, which implies that Stripe nodes follow the document order. However, due to subtree sharing, shared Path Stripe node ordering on their *start* value does not always reflect document order for the tree nodes being shared. We discuss this issue in Section 5.5.1.

For storing shared nodes in Path Stripes while being able to reconstruct the original tree nodes, we need extra sharing information; the *trace* value. To provide a compact representation of Path Stripe nodes, we decided the following:

- A shared node's *mult* value is not explicitly stored, as it can be computed directly from the *trace* value; it is the product of all node multiplicity values m_i.
- Any kind of sharing information, such as *trace* is not required for storing non-shared nodes ($mult = 1$ and $trace = \{\}$). Thus, we merely store this extra information for shared nodes. To accommodate this, we encode Path Stripe nodes as variable-size tuples: $\langle start, end, par, trace \rangle$. As a result, even in the case where the overall compression ratio is low, we pay no extra storage cost, apart from the implicit cost for storing tuples of variable size.
- For an efficient, variable-size encoding of *trace*, we use an implementation of

Dewey decimal encoding [82].

- When computing the *trace* value for a shared node N of path p, we only store the $(m_i : e_i)$ pairs of the shared nodes corresponding to any of the prefix paths of p, plus the pair for N for keeping each original tree node's extent. As a result, the length ℓ of *trace* is constrained by the number of the shared ancestor nodes (plus one) and thus, is usually significantly smaller than the length of p.

5.3 Document Loading

We now describe the loading process of an XML tree T into our native store. The loading of Attribute and Value Stripes is performed as described in Section 4.3 and is therefore omitted from this section.

As we parse the input XML tree, we construct an auxiliary tree structure. Each of the auxiliary tree's nodes contains the $(start, end)$ values, the *mult* value, a list of pointers to its children nodes and a pointer to its previous sibling node in the auxiliary tree. For any such node, firstChild(), lastChild() return their first and last child nodes respectively, children() returns the set of all children nodes and bisimilar(n) identifies whether a node is bisimilar to another node n. Finally, isShared() checks whether the node corresponds to a shared node while removeSubtree(n) removes its subtree rooted at its child node n.

The basic idea of the loading process is that it constructs a tree, say T', and identifies bisimilar nodes in a bottom-up fashion. Each node of T' is then stored as a Stripe node in the appropriated Path Stripe, indicated by path p. Whenever two sibling nodes are found to be bisimilar, the subtrees rooted at those nodes are shared in a single instance (Algorithm 5.3, lines 4-5). Subtree sharing results in updating recursively (through function merge) all nodes of the left subtree to reflect the sharing process (Algorithm 5.5, line 2) and the removal of the right subtree (Algorithm 5.5, line 3). Then, the multiplicity of the shared instance is updated accordingly (Algorithm 5.5, line 4).

Algorithms 5.1 and 5.2 correspond to the SAX events that signal the start and end of an element. When an element is identified, a new auxiliary tree node is created for that element and its *start* value is assigned (Algorithm 5.1, line 2). The node is then added to the tree structure through function addNode (Algorithm 5.1, line 3) where the pointer to the previous sibling node is set accordingly (Algorithm 5.4, line 4). On

Algorithm 5.1: startElement(*name*)

Result: Construct new element node with tag name *name*

```
1 begin
2     node ← createNode(name);
3     addNode(node);
4     st.push(node);
5 end
```

Algorithm 5.2: endElement()

Result: Update node

```
1 begin
2     current ← st.pop();
3     store ← updateNode(current);
4     if (st.size() = 1 && store = true) then
5         storeSubtree(st.top().firstChild(), {});
6     else if (st.size() == 0) then
7         storeSubtree(current, {});
8 end
```

Algorithm 5.3: updateNode(*child*)

Result: Update node

```
1 begin
2     updateStripeNode(node);
3     if (node.prevSibling = ⊥) then return false;
4     else if (node.bisimilar(node.prevSibling)) then
5         mergeSubtrees(st.top(), node.prevSibling, node);
6         return false;
7     return true;
8 end
```

Figure 5.2: Loading process (a)

encountering the closing tag of an element, the *end* and *par* values of the node are assigned (Algorithm 5.2, line 3). At that point we choose not to store the auxiliary tree node (as a Stripe node) as if a future sharing occurs affecting that node, we will need to update the Stripe contents to reflect the node sharing. This is certainly an expensive

Algorithm 5.4: addNode($node$)

Result: Add node n

```
1 begin
2     if (st.size() = 0) then return ;
3     par ← st.top();
4     n.prevSibling ← par.lastChild();
5     par.addChild(n);
6 end
```

Algorithm 5.5: mergeSubtrees($node_1, node_2$)

Result: Merge subtrees rooted at $node_1$, $node_2$

```
1 begin
2     merge(node1, node2);
3     node1.parent.removeSubtree(node2);
4     node1.mult++;
5 end
```

Algorithm 5.6: merge($node_1, node_2$)

Result: Merge subtrees rooted at $node_1$, $node_2$

```
1 begin
2     (C1, C2) ← (node1.children(), node2.children());
3     foreach (c1, c2) ∈ (C1, C2) do
4         merge(c1, c2);
5     updateStripeNode(node1, node2);
6 end
```

Algorithm 5.7: storeSubtree($node, trace$)

Result: Store all nodes of the $node$ subtree to Path Stripes

```
1 begin
2     if (node.isShared()) then
3         trace ← trace ∪ {node.mult : node.ext};
4     storeStripeNode(node, trace);
5     foreach n ∈ node.children() do
6         storeSubtree(n, trace);
7 end
```

Figure 5.3: Loading process (b)

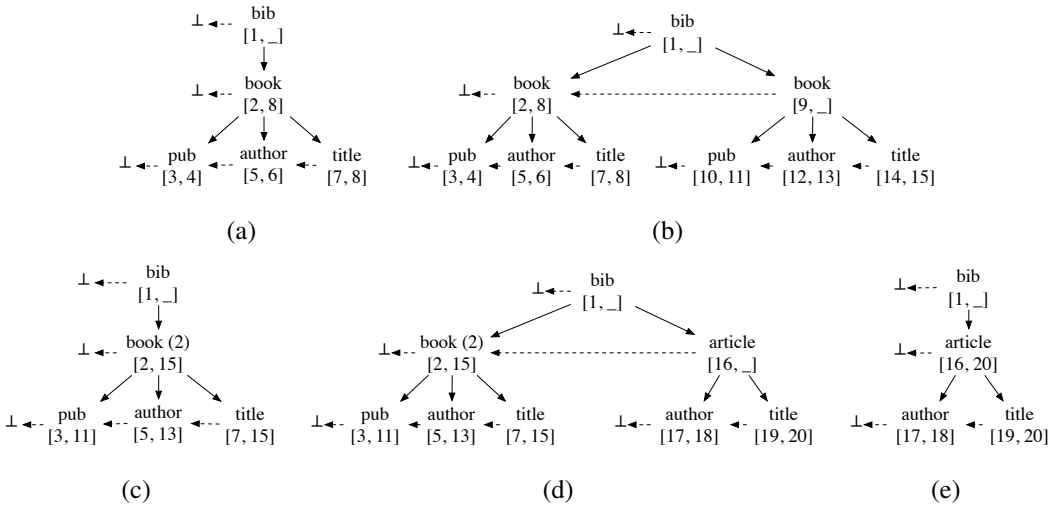


Figure 5.4: XML loading example

operation especially if we consider that we share subtree structures and therefore we would have to update all Stripe nodes corresponding to all nodes of a subtree. Therefore, we employ a lazy striping process, keeping at most two (shared) subtrees of the root element of T' in memory before actually striping them, (Algorithm 5.2, lines 4-5), through function storeSubtree. The downside of being lazy is that in the worst case $|T'|$ is equal to $|T|$ (*i.e.,* no sharing occurs and all elements are descendants of the only two children of the root). As our input trees can be arbitrarily large, we store the auxiliary tree structure as a memory mapped file, delegating paging responsibility to the operating system.

Example 5.3: We now describe five snapshots of the loading process, as these are depicted in Figure 5.4 for a part of the document of Figure 5.1(a). The tree of snapshot 5.4(a) corresponds to T' after calling endElement() for the first `book` node of T. The subtree rooted at the `book` node of T' is not yet striped. The state of T' when updateNode() in endElement() is called for the second `book` node is depicted in snapshot 5.4(b). The new `book` node is assigned its *end* and *par* values[1]. As its previous sibling is a bisimilar node, the two subtrees rooted at these nodes are being collapsed, resulting in the tree of snapshot 5.4(c). In snapshot 5.4(d), T' is depicted when the first `article` node ends. At that point, the `article` node is not bisimilar to the `book` node and since they are both the root's children, we store the `book` subtree resulting in the tree depicted in snapshot 5.4(e). Note that tree T' can contain shared subtrees, which

[1]We only show the $(start, end)$ pair for brevity.

in turn may contain shared subtrees due to bisimilar nodes at a different level. Such an example occurs in the last two `article` nodes of T which contain two consecutive `author` nodes. □

5.4 Node Reconstruction

Given the information stored in a shared node N of a Path Stripe, it is possible to decompress N and "split" it back into $n_0, ..., n_{mult-1}$ tree nodes. To identify any n_k's *start* position, we need to skip all document nodes from n_0 to n_k. Recall that we store this information in the extents of $trace$. In addition, note that the extent of each n_k is e_ℓ where ℓ is the length (*i.e.,* the number of $(m_i : e_i)$ pairs) of $N.trace$. This means that we only need to compute $n_k.start$ since $n_k.end = n_k.start + e_\ell$. The algorithm for splitting a shared node is shown in Algorithm 5.8. The intuition is that starting from the *start* value of the shared node (*i.e.,* the *start* value of the first tree node that is shared) we use the shared ancestor node extents to "skip" document nodes until we identify the requested tree node: Starting from the *start* value of the shared node (line 3), we traverse $trace$ backwards (line 5). For each $(m_i : e_i)$ pair, we increment the *start* value by the appropriate extent value (line 6), until the requested tree node's *start* is identified (line 8). We then update the rest of the node's structural information (lines 9-11) and exit the algorithm.

Example 5.4: The second Stripe node in the `/bib/article/author` Path Stripe, shown in bold in Figure 5.1(d), represents four tree nodes: $n_0, \ldots, n_3$ (also shown in bold in Figure 5.1(a). The shared node's $trace$: {2:6/2:1} has two pairs, denoting two `/bib/article` ancestor shared tree nodes, each containing two `/bib/article/author` tree nodes. The algorithm iterates over the $trace$ pairs; for each pair, it computes an offset which is used to increment the shared node's *start* value. In addition, j in Algorithm 5.8 is divided by the number of shared nodes for that pair until a value of 0 is reached, which means that we have identified the requested tree node. Function

$$\mathsf{splitNode}(\langle(22,32),21,4,\{2:6/2:1\}\rangle,3) \text{ returns node } n_3 = \langle(31,32),28,1,\{\}\rangle$$

□

As in the general case, we use the Stripe abstraction to emulate XML node sequences during query evaluation (Section 3.2.1). However, we depart from the generic

Algorithm 5.8: splitNode

Data: Path Stripe node: N, int: k

Result: Split Stripe node N having trace of length ℓ and and return its k_{th} tree node occurrence, n_k

```
1  begin
2      par_offset ← calcalcParentOffsets(N);
3      n_k.start ← N.start;
4      j ← k;
5      for i ← ℓ downto 1 do
6          n_k.start ← n_k.start + (j mod m_i) · (e_i + 1);
7          j ← ⌊j/m_i⌋;
8          if (j = 0) then break ;
9      n_k.end ← n_k.start + e_ℓ;
10     n_k.par ← n_k.start − par_offset[k mod m_ℓ];
11     n_k.⟨sh_start, sh_end, sh_par⟩ ← N.⟨start, end, par⟩;
12     return n_k;
13 end
```

Algorithm 5.9: calcParentOffsets

Data: Path Stripe node: N

Result: Return parent node offsets for all shared tree nodes in N

```
1  begin
2      m ← m_ℓ;
3      par_offset[0] ← N.start − N.par;
4      for i ← 1 to ℓ do
5          par_offset[i] ← par_offset[i − 1] + e_i + 1;
6      return par_offset;
7  end
```

Figure 5.5: Node Reconstruction

encoding of tree nodes by adding the triplet $\langle sh\_start, sh\_end, sh\_par\rangle$ and the new encoding for XML tree nodes thus becomes:

$$\langle start, end, par, level, kind, label, text, sh\_start, sh\_end, sh\_par\rangle$$

The new attributes are the *start*, *end* and *par* values of the compressed nodes. Note that these attributes are not needed to represent a tree node. However, as it will be explained in Section 5.5, such information is necessary for retrieving shared Path Stripe nodes. The construction of XML tree nodes from Stripe nodes at Stripe Scan operators generally follows the same logic as in the case of the explicit storage scheme. The new added attributes, basically involve only Path Stripes. When decompressing a shared Path node, its $\langle start, end, par \rangle$ triplet is copied to the $\langle sh\_start, sh\_end, sh\_par \rangle$ triplet respectively (Algorithm 5.8, line 11), while the tree node structural information is set as described above. In the case of Attribute or Value Stripes, since they do not involve sharing, the $\langle sh\_start, sh\_end, sh\_par \rangle$ triplet contains the same values as the $\langle start, end, par \rangle$ triplet.

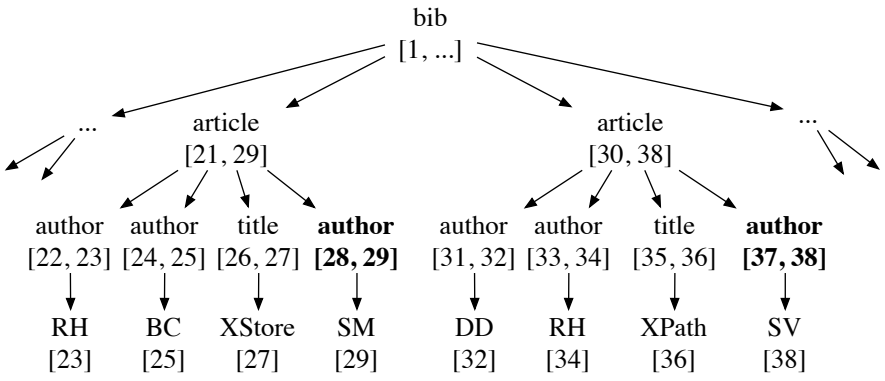
5.5 Query Evaluation Implementation Details

We now describe the challenges presented by the Tree-Sharing compression storage scheme during query evaluation.

5.5.1 Stripe Scans

As already described in Section 5.2, all *shared nodes* in a Path Stripe are stored in ascending order on their *start* value, *i.e.,* the *start* value of the first of the tree nodes being shared for each Stripe node. When we retrieve all shared nodes in *start* ascending order and split them to reconstruct the original tree nodes, it is usually the case that we retrieve the latter in document order. Consider, for instance, Path Stripe `/bib/article/author`, which contains two shared nodes as depicted in Figure 5.1(d). Once we retrieve the first shared node $N_1 : (17,18)$[2] which is in fact a non-shared Stripe node, we get tree node $n_1 : (17,18)$. Since no other tree node is shared in N_1, it is discarded and we retrieve next shared node $N_2 : (22,32)$. This shares four tree nodes as its *mult* value indicates. Reconstructing each one in sequence, we retrieve tree nodes $n_2 : (22,23)$, $n_3 : (24,25)$, $n_4 : (29,30)$ and $n_5 : (31,32)$. The order in which we retrieved the original XML tree nodes is the document order and is easy to verify that we effectively reconstructed the tree nodes stored in the corresponding Path Stripe of the explicit storage scheme, as this is shown in Figure 5.1(c).

[2]We denote nodes using their $(start, end)$ pair for brevity.



(a) An XML document

/bib/article

[s,e]	par
[...]	...
[21,27]	1
[28,34]	1
[...]	...

/bib/article/author

[s,e]	par
[...]	...
[22,23]	21
[24,25]	21
[28,29]	**21**
[31,32]	30
[33,34]	30
[37,38]	**30**
[...]	...

(b) A subset of (explicit) Path Stripes for the document of Figure 5.6(a)

/bib/article

[s,e]	par	mult	trace
[...]	...	...	{...}
[21,38]	1	2	{2:8}
[...]	...	...	{...}

/bib/article/author

[s,e]	par	mult	trace
[...]	...	...	{...}
[22,34]	21	4	{2:8/2:1}
[28,38]	**21**	**2**	**{2:8/1:1}**
[...]	...	...	{...}

(c) A subset of (compressed) Path Stripes for the document of Figure 5.6(a)

Figure 5.6: An example case of interleaved shared Stripe nodes

Nevertheless, retrieving Path Stripe nodes ordered by *start* ascending order cannot always guarantee that the reconstructed tree nodes will be produced in document order. Consider the following example: Figure 5.6(a) depicts a fragment of the XML tree of Figure 5.1(a), with two new `author` elements added, one for each `article` element. During the document loading, the two subtrees rooted on `article` element nodes are shared and a single Path Stripe node is inserted to `/bib/article` Path Stripe having a *mult* value equal to two. As for their `author` children nodes though, since the two `article` nodes are now shared, there will be two Stripe nodes created for `/bib/article/author` Path Stripe; the first Stripe node will contain the first two `author` children nodes of each `article` node, *i.e.,* four nodes in total, while the sec-

ond one will contain the last `author` child node of each `article` node, two in total. This is depicted in Figure 5.6, where the two new `author` nodes are illustrated in bold characters in the original XML tree (Figure 5.6(a)) as well as at the Path Stripe node that shares them (Figure 5.6(c)). During node retrieval now, the first Stripe node, will reconstruct the first two `author` children nodes of each `article` node, *i.e.,* nodes (22,23), (24,25), (31,32) and (33,34), while the second Stripe node will reconstruct nodes (28,29) and (37,38). Clearly, the produced tree nodes do not satisfy document order.

In order to satisfy one of the fundamental properties of the Stripe Scan operators, *i.e.,* to produce nodes in document order, we had to change the Stripe Scan operators implementation in order to account for cases as the one described above. This particular problem arises when there are nodes of the same label (same label-path in particular) that are not shared, while a sharing occurs between at least one of their ancestor nodes. In such cases, we say that the produced Path Stripe nodes are *interleaved*, *i.e.,* the *start* value of the second Path Stripe node occurs before the *end* value of the previous Path Stripe node. Stemming from this observation, we provided the Stripe Scan operator with a tree node cache and modified its next() method (when Path Stripes are involved) to operate as follows:

1. When a tree node is requested from a Stripe Scan operator, a Path Stripe node is retrieved.

2. If the retrieved node is not shared, *i.e.,* it merely contains a single tree node, we simply return it.

3. If the retrieved node is a shared one, we continue retrieving Path Stripe nodes until one is found that is not interleaved with respect to its previous retrieved node.

4. We then reconstruct the first tree node of each of the retrieved Path Stripe nodes, merge them and output the one with the minimum *start* value. During the next next() invocation, the Path Stripe node which contained the tree node that was lastly produced is asked to reconstruct its next tree node and the process continues in this manner until all retrieved Path Stripes are exhausted, *i.e.,* all their tree nodes are reconstructed.

Example 5.5: Let $N_1 : (22, 34)$ and $N_2 : (28, 38)$ be the two shared nodes for Path Stripe `/bib/book/author`. During the first invocation of next(), N_1 is retrieved and since it

is a shared node, we cache it and continue retrieving nodes until we reach a node that is not interleaved in its previous node. As a result, node N_2 is fetched and cached as well. Now, we reconstruct the first tree nodes of both Path Stripe nodes N_1, N_2, resulting in tree nodes $n_1 : (22,23)$ and $n_2 : (28,29)$ and we produce node n_1. During the next invocation of next(), since there are cached Stripe nodes, we reconstruct the next tree node from N_1 that contained node n_1, previously produced. Tree node $n_3 : (24,25)$ is produced and between n_2 and n_3, tree node n_3 is returned to the caller operator. The next time, shared node N_1 will reconstruct node $n_4 : (31,32)$ and tree node n_2 produced from shared node N_2 will be produced to the caller operator. The process continues likewise until all tree nodes are produced: $n_4 : (31,32)$ $n_6 : (33,34)$ and $n_5 : (37,38)$. □

The size of the cache is in the worst case $O(fan\_out)$ where *fan_out* is the *maximum* fan out of the original XML tree. However, even this bound which is a reasonable upper bound for most XML documents is hard to reach since it implies that all elements have the same label and all of them have different tree structure so that they cannot be shared. This rarely occurs in real life XML documents.

5.5.2 Navigation and Shared Path Node Issues

In Section 3.3, we described that when a caller operator requests a node from its input operator(s), it can direct them by passing a "hint" node so that the desired node is retrieved. This is especially useful in navigation algorithms, such as structural joins and we have already described possible values of a hint node according a context node and the direction of the navigational operation which is defined by the axis parameter.

The compression of element tree nodes in shared Path Stripe nodes is a desired effect of the tree-sharing storage scheme. However, during query evaluation, it adds a certain degree of complexity. In particular, it becomes cumbersome to decide the appropriate hint node for the retrieval of candidate element nodes. We now discuss such issues and provide with the identification of the proper hint node that must be passed to input operators for producing the correct candidate nodes.

Suppose we have successfully retrieved a context node c and now aim to retrieve the *descendant* candidate nodes from an input operator. Recall that we aim to locate candidate nodes that occur in the candidate range $(c.start, c.end]$. The proposed hint node was a node having $hint.start = c.start + 1$ so that all descendant candidate nodes will be retrieved in sequence. However, if the candidate nodes are element nodes then

the lowest-level operator will be a Stripe Scan operator accessing Path Stripe nodes. Due to Stripe node sharing though, passing the hint node value proposed above, could result in loosing the requested candidate nodes. Consider the following example: Suppose that our context node is the `article` element node $c : (28, 34)$ in Figure 5.1(a). The requested candidate nodes are the ones that occur within candidate range $[29, 34]$ of *start* values. However, passing a hint node with the *start* value 29 will fail to retrieve any candidate nodes. This occurs because the context node is part of a shared Stripe node and its descendant nodes are also part of shared nodes located in the corresponding Path Stripes. Indeed, if we have a look at the `/bib/article/author` and `/bib/article/title` Path Stripes that contain its descendant candidate nodes, no Stripe node can be retrieved when passing the proposed hint node.

To generalise this observation, when context node c is part of a shared Stripe node, then we need to identify a proper hint value that will effectively identify any shared Path Stripe node that contains candidate nodes with respect to context node c. Therefore, we enrich the hint node argument adding the *sh_start* attribute. This value is considered by the Path Stripe Scan operators and used to retrieve shared Stripe nodes that contain candidate nodes; the *start* value of the hint is then used for locating the appropriate tree candidate nodes that are reconstructed by the retrieved shared Stripe nodes.

We now consider which is the appropriate *sh_start* value that must be set for the hint node for retrieving candidate nodes with respect to a given context node c for a certain axis parameter. In fact we strive to identify the minimum $hint.sh\_start$ that must be used so that the Stripe Scans will retrieve shared nodes that contain candidate nodes.

When the *descendant* axis is considered, the minimum value for $hint.sh\_start$ must be $c.sh\_start + 1$, *i.e.,* the *start* value of the shared node that contains context node c. Indeed, to locate the descendant candidate nodes of context node c in the above example, the hint node must have $hint.sh\_start = 22$, which is the *start* value of the Stripe node (plus 1) which reconstructed context node c. The Stripe Scan operators will use that value to retrieve the shared Stripe nodes that may contain candidate nodes; value $hint.start = 29$ is now used for producing the reconstructed candidate tree nodes that are descendants of the context node c. The same holds when considering the *child* axis.

Similar observations apply when considering the *following-sibling* axis. Passing a hint node with $hint.start = c.end + 1$ as the candidate range $(c.end, c.par\_end)$ in-

structs, will fail in retrieving the corresponding candidate nodes when context node c is part of a shared Path Stripe node. To correctly identify all following-sibling candidate nodes, the hint node must be passed with $hint.sh\_start = c.sh\_start$ and $hint.start = c.end + 1$, so that the appropriate shared node is retrieved from the Stripe Scan operator using $hint.sh\_start$ value and then the sibling tree nodes are identified using $hint.start$ value. For instance, suppose that our context node is the `author` element node $c : (29, 30)$ in Figure 5.1(a). The requested candidate nodes are the ones that occur within the candidate range $[31, 34]$ of *start* values. To retrieve all following-sibling candidate nodes, we must use $hint.sh\_start = c.sh\_start = 22$ and $hint.start = c.end + 1 = 31$. Thus, the shared Path Stripe node (22,32) will be retrieved containing the following-sibling `author` node (31,32).

Likewise, for retrieving *following* candidate nodes with respect to a context node c, we need to determine the minimum $hint.sh\_start$ value to correctly retrieve the shared Path Stripe nodes that contain the requested candidate tree nodes. The candidate range is now $(c.end, r.end)$, where r is the document root node. This effectively means that candidate tree nodes will also include nodes that are siblings of any of the context node's ancestor nodes, including itself. Thus, the minimum *sh_start* value is the *start* value of the top most shared Stripe node among the path from the root of the document to the context node. As an example, consider that our context node is the `author` element node $c : (24, 25)$ in Figure 5.1(a). The range for following candidate nodes with respect to context node c is $[26, 33]$. A first observation is that in order to retrieve following `author` nodes (29,30), (31,32), the minimum value for $hint.sh\_start$ that should be used is the context node's *sh_start* value since these nodes are shared in the same Path Stripe node as context node c. However, using that value for the hint node would fail to retrieve the `article` node (28,34) that also follows context node c. This is because this node is shared along with the parent node of the context node and thus the minimum value for $hint.sh\_start$ in order to retrieve all candidate nodes is *sh_start* of c's parent node. Generalising this observation, the value for $hint.sh\_start$ must be the *start* value of the top-most ancestor node of the context node that is being shared.

Note that the hint value must be computed as described above, only when the context node is part of shared Path Stripe node. If this does not hold, then the hint node is computed as described in Section 3.3; the $hint.start$ value is set and then copied to $hint.sh\_start$ value to be later used by the Stripe Scan operators. This way, the evaluation of structural join algorithms remains in a large part unaffected even in the case where the XML document structure is poorly compressed.

5.6 Experimental Results

We now present our experimental results regarding the proposed tree-sharing compression storage scheme. Our experimental setup, *i.e.,* hardware platform, software tools, XML datasets and tested queries, is the same as described in Section 4.5. The query engine of our native store prototype is largely unaffected by the underlying storage scheme; we merely needed to change the Scan Operators, to accommodate compressed Path Stripe nodes retrieval and resolve ordering element node issues as presented in Section 5.5.1. In addition, to resolve issues concerning locating shared nodes, as discussed in Section 5.5.2, we also performed minor changes to our structural join evaluation algorithms; nevertheless, the core functionality of the evaluation algorithms remained unchanged.

Our experimental study is again divided in three parts, one for each of the Xmark, Mbench and DBLP datasets. We shredded all XML documents in our native store using the tree-sharing compression storage scheme. To demonstrate the benefits and/or drawbacks of tree-sharing compression, we directly compare our results to those produced using the explicit (uncompressed) storage scheme, as presented in Chapter 4. To disambiguate our query engines, our native XML store over the explicit storage scheme is referenced as SRX, while CSRX stands for our native XML store over the compression storage scheme. During query evaluation, we have enabled all possible optimisations and thus both query engines operate using the PRO evaluation setup.

5.6.1 Compression Effectiveness

To measure the compression effectiveness of the proposed tree-sharing compression, we define compression ratios in terms of both Path Stripe nodes being compressed (shared), CR_n, as well as Path Stripe storage size (I/O pages) CR_p. The *node compression ratio*, CR_n is defined as the fraction of Path Stripe nodes in SRX to the Path Stripe nodes in CSRX, *i.e.,* $CR_n = \frac{N_{SRX}}{N_{CSRX}}$. In essence, the node compression ratio is the average multiplicity of all Path Stripe nodes in CSRX. Similarly, the *size compression ratio*, CR_p is defined as the fraction of Path Stripe pages (that is, the number of I/O pages needed for Path Stripe storage) in SRX to the Path Stripe pages in CSRX: $CR_p = \frac{P_{SRX}}{P_{CSRX}}$. Tree-sharing compression results in Stripes having at most the same number of nodes of the corresponding Stripes in the explicit storage scheme. Hereafter, we use the terms "node reduction" and "size reduction" to denote the compression impact on Stripes in

Dataset	CR_n	CR_p
Xmark100	1.34	1.32
Mbench10	3.95	3.46
DBLP10	16.92	11.17

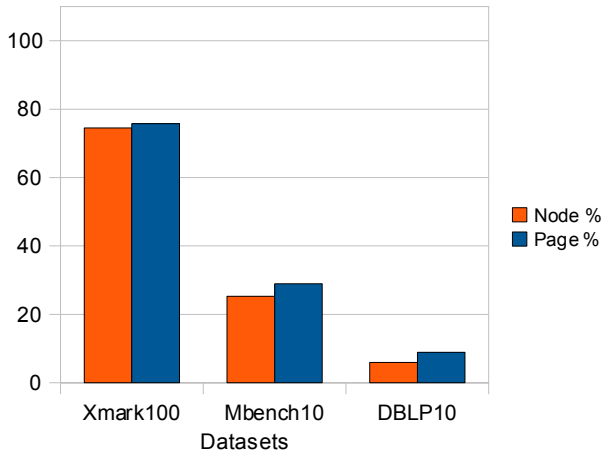


Figure 5.7: Compression effectiveness overview

terms of the number of Stripe nodes and the number of Stripe pages being reduced, respectively.

We now provide an overview of the compression effectiveness for all three datasets: Xmark, Mbench and DBLP. The compression ratios are presented in Figure 5.7, for the largest of each of the XML datasets. Our proposed compression technique performs rather poorly on the Xmark dataset, with node compression ratio being relatively close to 1:1; 1.34 to be precise. In essence, this means that Path node sharing seldomly occurs for the Xmark dataset and that the majority of the Path Stripe nodes are the same as in the explicit storage scheme. The compressed Path Stripes of CSRX are reduced by 25% in both nodes and pages. For the Mbench dataset, however, the merits of structural compression become evident; the Path node compression ratio is close to 4:1, implying that each CSRX Path Stripe node is equivalent to four SRX Path Stripe nodes. The Path size compression ratio is close to 3.5:1, resulting in size reduction of more than 70%, compared to the size of Path Stripes in the explicit storage scheme. The impact of tree-sharing compression is even clearer for the DBLP dataset. The achieved node compression ratio is close to 17:1, implying significant Path Stripe node sharing. This is also reflected to the size required for the compressed Path Stripes storage; less than 10% of the original Path Stripe size.

The node and size compression ratios provide the impact of our proposed compression technique on Path Stripe storage. This also impacts query performance since reducing the query input (Stripe) size usually results in better evaluation times. However, Stripe size reduction cannot always guarantee faster query evaluation. There are

a number of reasons for that of which the most significant are:

The query evaluation algorithms Depending on the query semantics, a range of evaluation algorithms and access methods are used that may consume all or parts of the query input. If an access method, for instance, only accesses a small part of a Stripe, then reducing the Stripe size may have no impact or even hurt query performance. Nevertheless, a sizeable Stripe size reduction due to node compression usually results in reducing I/O cost which is also reflected on response time.

The added decompression cost When compressed Path Stripe nodes are retrieved, the decompression process will generate the original Path Stripe nodes being shared. This imposes an extra computational cost, in comparison to SRX. Thus, to benefit from node compression, the I/O cost gain due to compression must make up for the computational cost of decompressing the shared Path Stripe nodes. Of course, this is merely a rule of thumb, as other conditions also influence query evaluation performance. For instance, buffer pool utilisation. Due to node compression, more element nodes can be buffered compared to SRX using the same buffer pool size. The downside is that we still need to pay the decompression cost for re-using a Path Stripe node, even if that resides on the buffer pool. Nevertheless, the same rule of thumb holds. Keeping more data in the buffer pool can effectively decrease the I/O cost as fewer pages will be paged out and written to disk. Thus, as long as the total decompression time is less than the time required to perform I/O, CSRX will benefit and perform better than SRX.

5.6.2 Xmark

In this section, we take a closer look on the impact of our proposed compression technique on the Xmark dataset. In Table 5.1, we provide details regarding Path Stripe nodes when both explicit and tree-sharing compression storage schemes are used. Note that the node compression ratio is constant regardless of the size of the Xmark dataset. We also provide the node savings, that is, the percentage of node reduction caused by node sharing. For the Xmark dataset, a 25% reduction in Path Stripe nodes occurs, that remains constant as the size of the dataset scales. In addition to Path Stripe nodes, we provide similar data regarding the I/O pages required for Path Stripe storage, presented in Table 5.2. The size compression ratio CR_p is also (almost) immune to dataset size, which is also reflected on I/O page savings, the percentage of page reduction

Xmark Scaling Factor	Path Stripe cardinality (Nodes)		CR_n	Node Savings
	SRX	CSRX		
0.1	167866	124379	1.35	25.91%
1	1666316	1244470	1.34	25.32%
10	16703211	12454322	1.34	25.44%
100	167095845	124559662	1.34	25.46%

Table 5.1: Compression effect on Path Stripe cardinality for Xmark datasets

Xmark Scaling Factor	Path Stripe size (Pages)		CR_p	Page Savings
	SRX	CSRX		
0.1	1526	1287	1.19	15.66%
1	10600	8189	1.29	22.75%
10	100625	76420	1.32	24.05%
100	999855	757766	1.32	24.21%

Table 5.2: Compression effect on Path Stripe size for Xmark datasets

accomplished when the compression storage scheme is used compared to the explicit scheme.

As already presented, the tree-sharing compression technique performs rather poorly on the Xmark dataset. Node compression ratios indicate that Path node sharing has minimum impact on the Xmark dataset, despite the fact that it is a quite regular XML dataset; one would expect node sharing to occur more frequently. In order to understand the cause of poor compression, we investigate the Xmark dataset's tree structure. Despite the dataset's structural regularity on most of its core elements, it also contains frequent occurrences of deeply-nested elements that enclose natural language text which, in turn, enclose HTML-like formatting markup instructions [88]. Such markup, as expected, occurs in a non-regular manner resulting in mixed content tree nodes and element subtrees with largely varying structure. This effectively reduces any chance for node sharing under the tree-sharing compression technique, which also propagates to the ancestor elements, as the subtree sharing is applied in a bottom up manner to XML trees. This results in an average node sharing (node compression ratio) very close to 1:1, indicating that node sharing hardly occurs in the Xmark dataset. This is also reflected on the storage size of the Path Stripes.

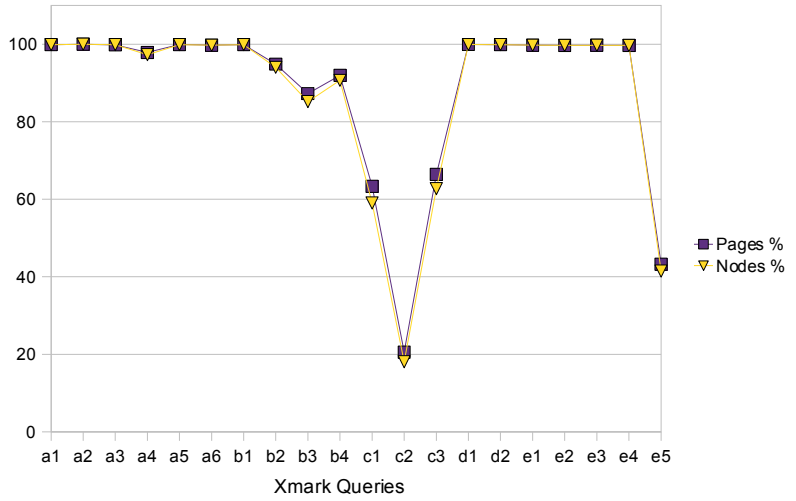


Figure 5.8: CSRX compression effect for Xmark queries

We now investigate the compression effect for each of the Xmark queries in isolation. We consider the Stripes that are required to evaluate each of the Xmark queries and compare their included nodes and storage requirements against those of the explicit storage scheme. This comparison is depicted in Figure 5.8, where the normalised Stripe nodes and I/O Stripe pages are displayed. This data corresponds to the largest of the Xmark datasets (sf=100, size=11GB). As seen from the graph, for most Xmark queries, the required Stripes that need to be accessed, contain almost the same number of Stripe nodes as when stored using the explicit storage scheme. On the other hand, for queries c1-c3 and e5, the required Stripe nodes are reduced by a large factor; there is a 40% node reduction for queries c1, c3, while for queries e5 and c2, the node reduction is around 60% and 80% respectively. Finally, for queries b2-b4, a small but sizeable node reduction between 6% and 15% is achieved. The same observation can be made for the size reduction achieved by the compression storage scheme for the Xmark queries. The size reduction is proportional to the node reduction. Note that all types of Stripes that are needed for the evaluation of each of the Xmark queries have been taken into account. However, the node and size reduction is an effect of Path Stripe compression.

We now proceed to the comparison of the query evaluation times of our native XML stores over the explicit and the compression storage scheme. The results for the largest of the Xmark datasets (sf=100, size=11GB) are displayed in Figure 5.9. The

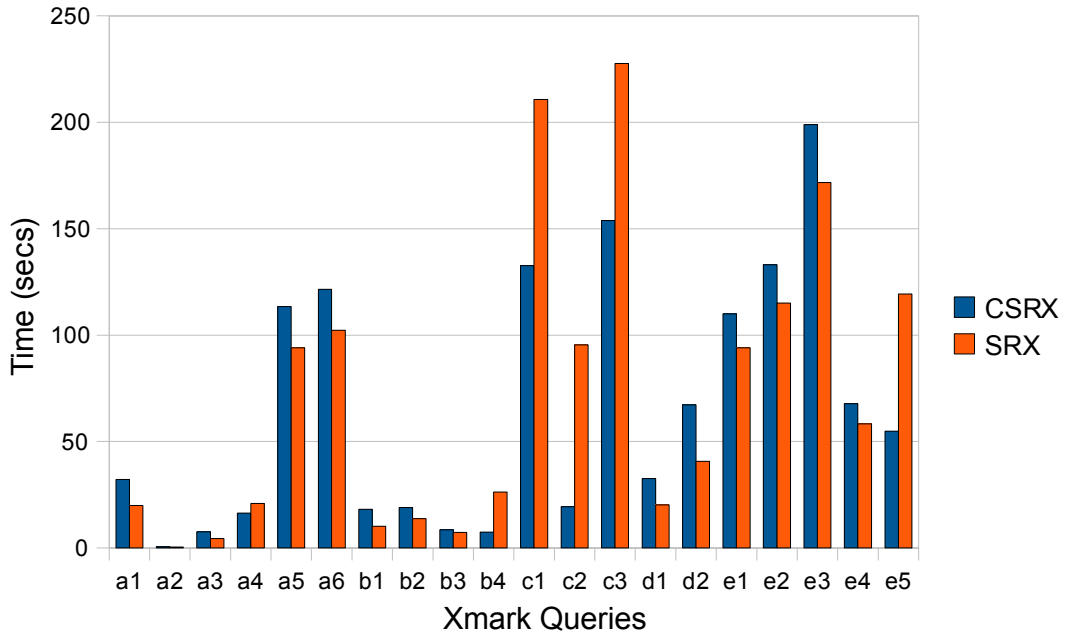
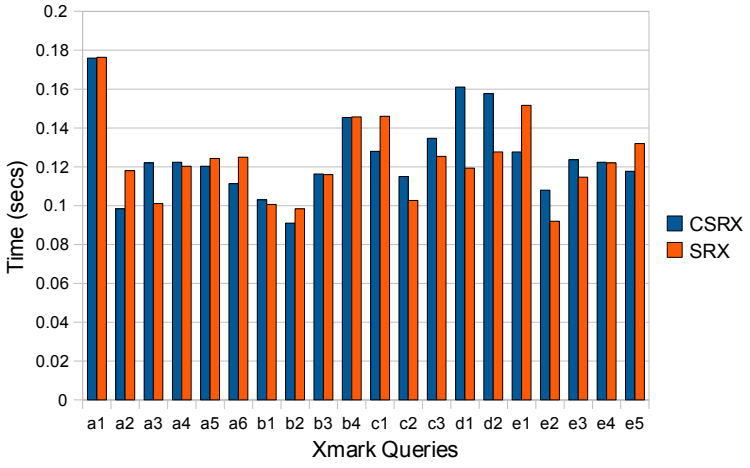


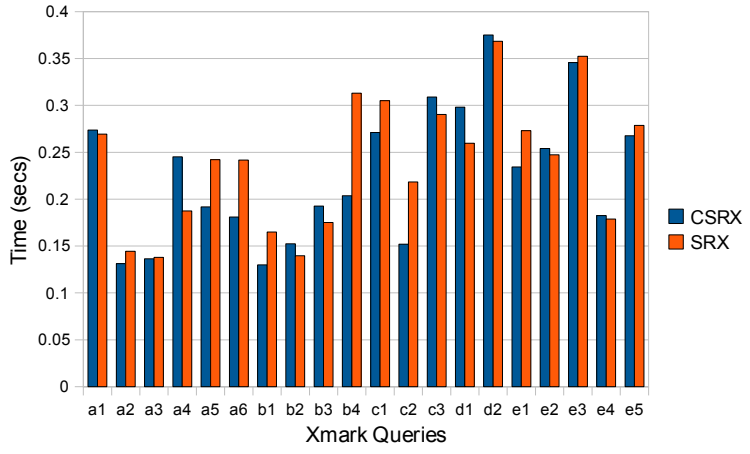
Figure 5.9: CSRX query evaluation for Xmark100 dataset

main outcome from this comparison is that for more than half of the Xmark queries, SRX outperforms CSRX. This is a rather expected result; the low node and size savings due to poor compression of the Xmark dataset combined with the added complexity of dealing with shared Path Stripe nodes, hurt overall query evaluation performance. On the other hand, the cases where CSRX is a clear winner over SRX are queries c1-c3 and e5. Recall that compression has a significant impact on the Stripes involved in such queries, reducing Path Stripe nodes and Path Stripe size by a large factor. For queries b2-b4, node compression results in a small size and node reduction; the results however look rather diverse: For query b2, although the Stripe size is reduced by 5%, SRX still performs better than CSRX. Similarly, for query b3, the size reduction of 12% seems to have no impact on query performance. Finally, for query b4, where the Stripe size is reduced by 8%, we notice an evaluation speedup of 71% in favour of CSRX. So why CSRX does not benefit from size reduction achieved for queries b2 and b3? The answer is that for both b2 and b3 queries, the size of the Stripes involved are relatively small, even when the explicit storage scheme is used. This, combined with the low achieved size reduction, results in relatively small I/O benefits that are either balanced (b3) or covered (b2) by the extra computational cost for decompression.

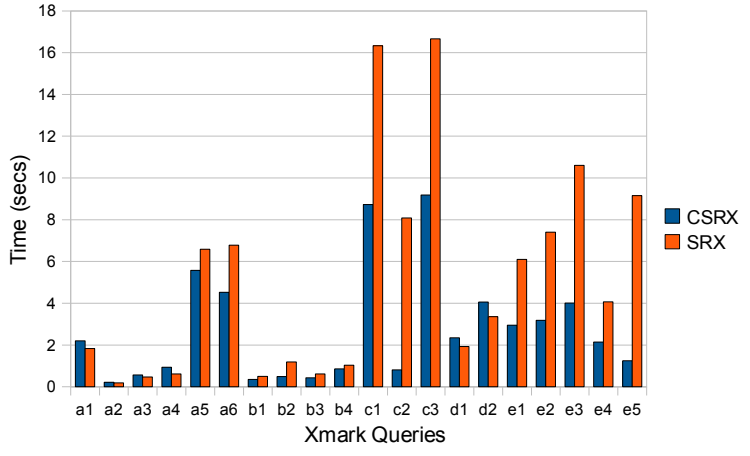
Regarding the compression effect on the smaller Xmark datasets, the evaluation results for scaling factors 0.1 to 10 are displayed in Figure 5.10. For the smaller of the



(a) Xmark0.1



(b) Xmark1



(c) Xmark10

Figure 5.10: CSRX query evaluation for Xmark0.1, Xmark1 and Xmark10 datasets

Mbench Scaling Factor	Path Stripe Cardinality (Nodes)		CR_n	Node Savings
	SRX	CSRX		
0.1	67697	19036	3.56	71.88%
1	738984	202202	3.65	72.64%
10	7291959	1845995	3.95	74.68%

Table 5.3: Compression effect on Path Stripe cardinality for Mbench datasets

Mbench Scaling Factor	Path Stripe size (Pages)		CR_p	Page Savings
	SRX	CSRX		
0.1	440	165	2.67	62.5%
1	4456	1412	3.16	68.31%
10	43635	12624	3.46	71.07%

Table 5.4: Compression effect on Path Stripe size for Mbench datasets

Xmark datasets, Xmark0.1, we observe that the CSRX and SRX evaluation results are in most cases comparable, while in certain cases, the poor compression favours SRX over CSRX. As the size of the dataset scales, we notice that for certain queries (*e.g.,* a5, a6) CSRX outperforms SRX. This is mostly evident in the results of the Xmark10 dataset and for queries a5, a6 and e1-e4. This is a slightly unexpected result as the compression impact on the input Stripes of these queries, is minimal. Looking closer look at the queries, they all involve (a) accessing `person` elements, and (b) predicate expressions on their sub-elements. The compression impact on the corresponding Stripes is indeed minimal for these queries, due to the high degree of irregularity present in the `person` subtrees. These results are a side-effect of the compression method on the input Stripes for this specific dataset. Although the input sizes for both storage schemes are essentially equal, a possible different distribution of nodes into Stripe pages may impact the number of pages scanned to evaluate the predicates. Recall that scan operators operating in filter mode do not necessarily access all Stripe nodes, since predicate evaluation is "short-circuited", *i.e.,* the expressions within predicates are not further evaluated once a truth value can be assigned. This, in conjunction with node caching by the scan operators, favours CSRX for the specific dataset.

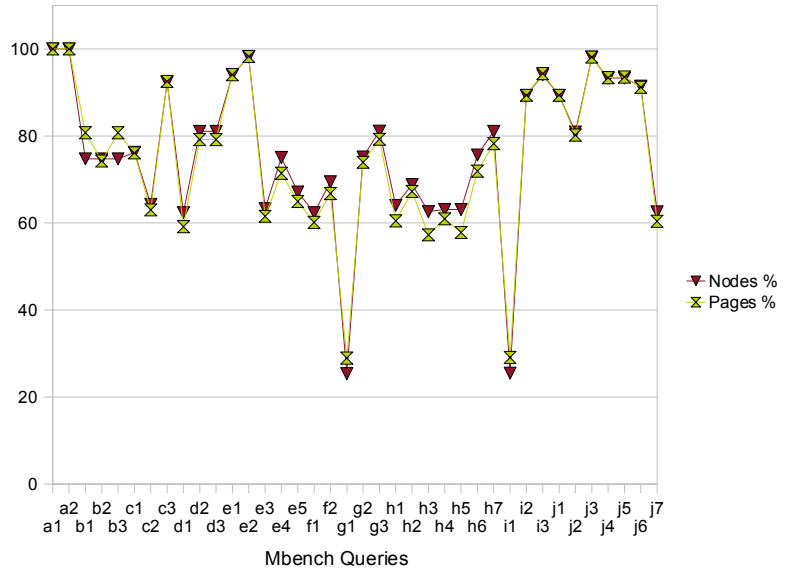


Figure 5.11: CSRX compression effect for Mbench queries

5.6.3 Mbench

We now examine the impact of our proposed compression technique on the Mbench dataset. In Table 5.3, we provide details regarding Path Stripe node compression accomplished by the compression storage scheme. At first glance, we observe that a significant node compression is achieved for the Mbench dataset, ranging from 3.5:1 to almost 4:1 for the largest dataset. Note that, in contrast to the Xmark dataset, the node compression ratio is increasing as the size of the dataset increases. The compression impact on the structure of the Mbench dataset becomes clearer when the node savings are considered. For the smallest of the Mbench datasets, Path Stripe nodes are reduced by 72%, while for the largest dataset, nodes are reduced by 75%. The compression impact on the Mbench dataset is also reflected on the Path Stripe storage requirements. As shown in Table 5.4, the achieved size compression is also significant, ranging from 3.7:1 to 3.5:1 for the largest of the Mbench datasets. Regarding I/O savings, the compression storage scheme enforces a reduction of 70% of the size required to store Path Stripes over the explicit storage scheme.

We now turn our attention to the compression effect for each of the Mbench queries, as depicted in Figure 5.11. Again, we present the normalised results with respect to the explicit storage scheme. The results correspond to the largest of the Mbench datasets

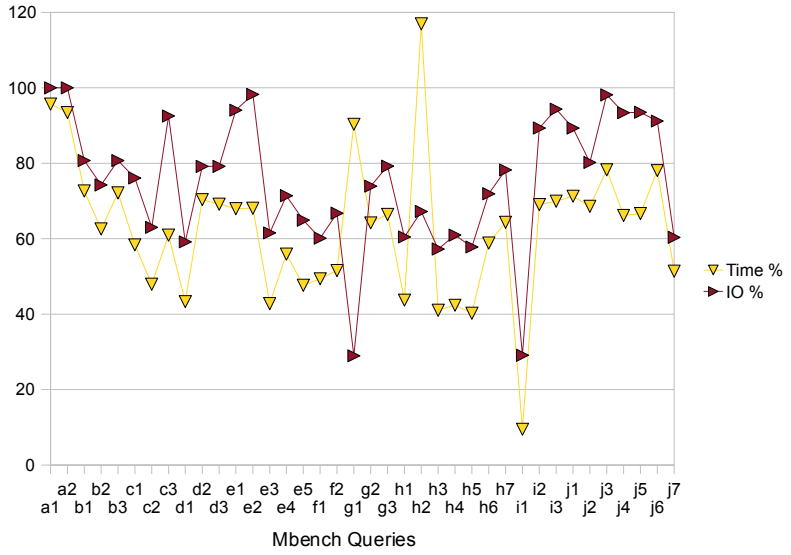
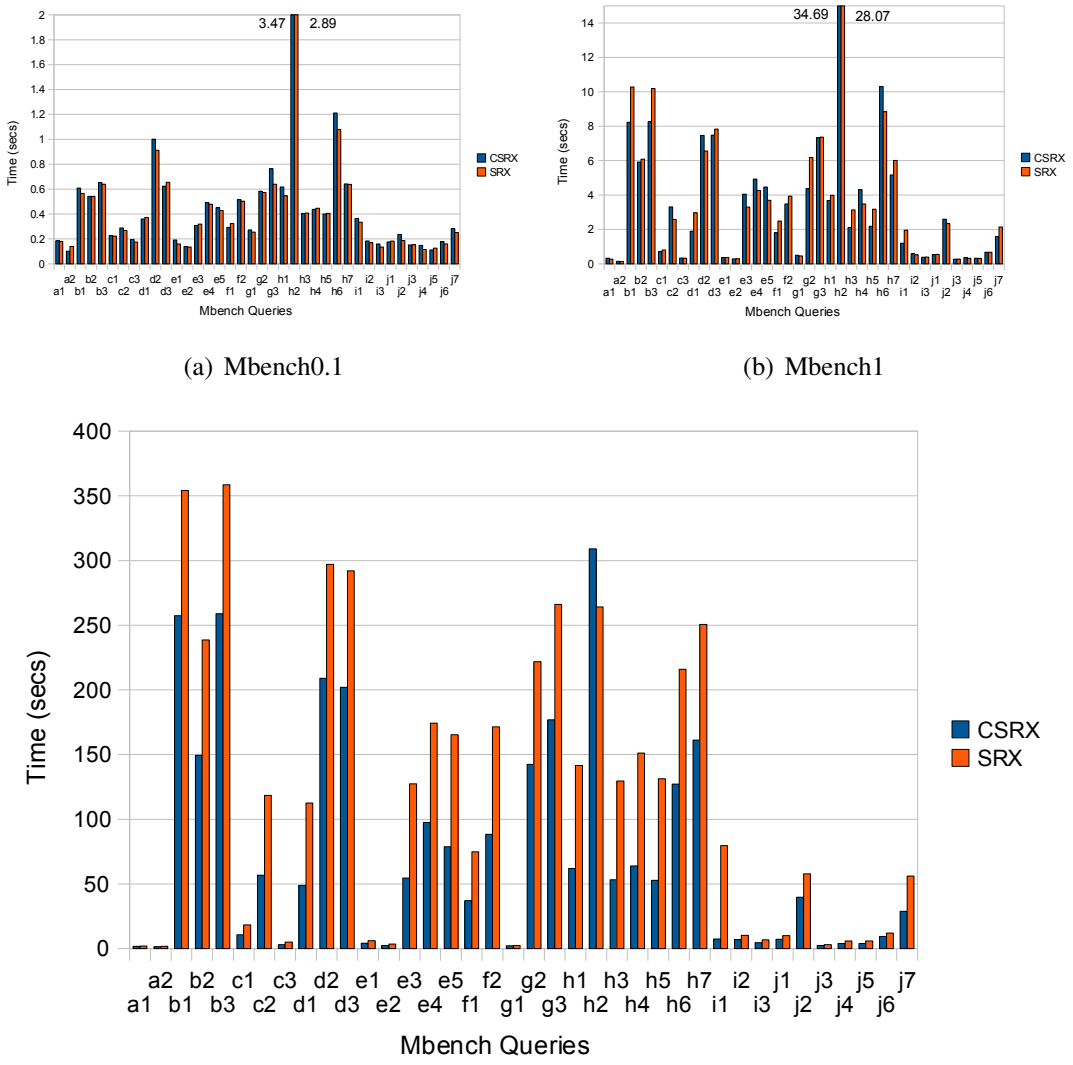


Figure 5.12: Evaluation time and I/O relationship for Mbench queries

(sf=10, size=5GB). For most of the Mbench queries, Path Stripe compression effectively contributes in reducing the number of Stripe nodes required for query evaluation. The average node reduction achieved is close to 24% and it leads to an average Stripe size reduction of 25% in comparison to the nodes and size required over the explicit storage scheme.

The node and size reduction affects query evaluation performance. The evaluation results of CSRX and SRX for the largest of the Mbench datasets, Mbench10, are displayed in Figure 5.13(c). As seen from the graph, for all queries (but one) of the Mbench Queries, CSRX outperforms SRX. CSRX achieves an average speedup of 37% compared to the evaluation times achieved over the explicit storage scheme. In fact, we observe that the evaluation time speedup in CSRX is, at most times, proportional to the achieved size reduction due to compression. This is illustrated in Figure 5.12 where the normalised evaluation time speedup and size reduction for Mbench queries are depicted.

As already explained, the measured Stripe size per query, is the total size of those Stripes that contain all nodes needed to formulate the query result. We have already discussed, that the reduction of Stripe size cannot always guarantee better evaluation times neither can always be proportional to the evaluation time speedup. For instance, consider query g1, where the accomplished size reduction (of 71%) is not reflected on the time speedup (of 10%). This is due to the following axis structural join operator,



(a) Mbench0.1

(b) Mbench1

(c) Mbench10

Figure 5.13: CSRX query evaluation for Mbench1, Mbench1 and Mbench10 datasets

employed for the evaluation of query g1, that merely uses a very small portion of its left input (see Section 3.3.2.3) and thus is less affected by the achieved size reduction (as far as its left input is concerned). Another example is query h2, where node compression has a negative impact on query evaluation, since the evaluation process requires a large number of Path nodes to be accessed. and thus decompressed, multiple times; the total decompression cost suppresses the I/O benefits caused due to node compression. Nevertheless, a sizeable Stripe size reduction due to node compression usually results in reducing I/O cost which is also reflected during query evaluation. This is evident in the vast majority of Mbench queries.

DBLP Scaling Factor	Path Stripe Cardinality (Nodes)		CR_n	Node Savings
	SRX	CSRX		
1	2609007	1542370	1.69	40.88%
5	13045027	1542370	8.46	88.18%
10	26090052	1542370	16.92	94.09%

Table 5.5: Compression effect on Path Stripe cardinality for DBLP datasets

DBLP Scaling Factor	Path Stripe Size (Pages)		CR_p	Page Savings
	SRX	CSRX		
1	15758	9828	1.6	37.63%
5	78153	13981	5.59	82.11%
10	156144	13981	11.17	91.05%

Table 5.6: Compression effect on Path Stripe size for DBLP datasets

Finally, we present our experimental results for the smaller Mbench datasets. The reported evaluation times of CSRX and SRX for the Mbench0.1 dataset are shown in Figure 5.13(a). We observe that despite the significant Path Stripe node and size compression accomplished (3.5 and 2.7 respectively), query evaluation is by a large factor unaffected and in fact for some queries, compression rather hurts query performance instead of boosting it. This occurs due to the fact that the actual size needed for Path Stripe storage is small, (1.9MB and 700KB for the explicit and compression storage schemes respectively) and thus the time spent on doing extra I/O operations at SRX is comparable to the time spent for decompressing shared nodes at CSRX. However, as the size of the Mbench dataset increases, so does the I/O cost for accessing relevant to the queries Stripes. Path Stripe size reduction due to compression now has a larger impact on query performance as it limits the dominant factor of query evaluation time, the I/O cost. For instance, in Figure 5.13(b), where we compare the evaluation times of CSRX and SRX for the Mbench1 dataset, we observe that for almost 1/4 of the Mbench queries CSRX performs better than SRX while for almost half of them, their evaluation times are comparable. However, when considering the largest Mbench dataset, Mbench10, the benefits of node compression are evident; CSRX improves evaluation times by 37% compared to SRX.

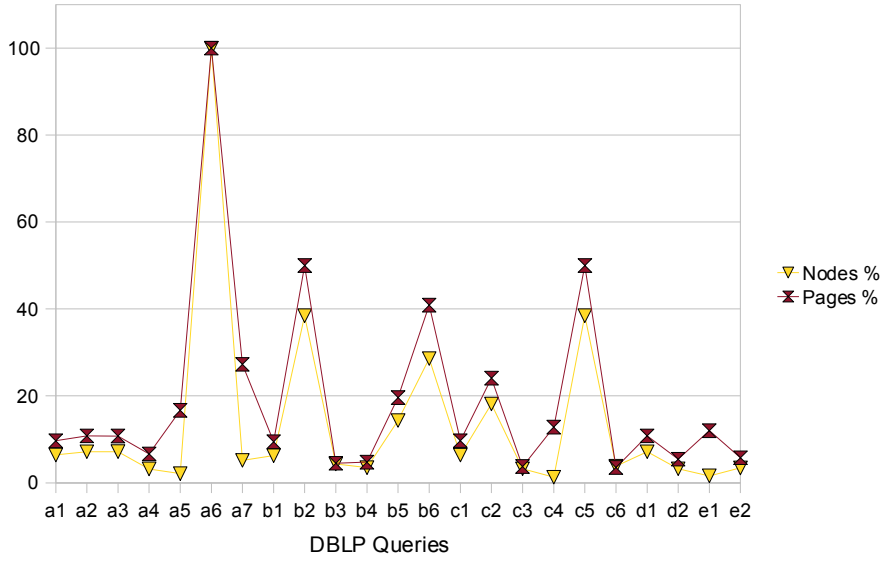


Figure 5.14: CSRX compression effect for DBLP queries

5.6.4 DBLP

We begin with the compression impact on the DBLP dataset. Details regarding Path Stripe node compression are displayed in Table 5.5. The node compression ratio for the smallest of DBLP dataset is relatively low; 1.7:1. However, Path Stripe nodes are reduced by 40% compared to the explicit storage scheme. As the dataset scales up and since the same document structure is copied multiple times, the achieved node compression ratio is increasing proportionally to the scaling factor; for the DBLP dataset of scaling factor 5 (DBLP5), the node compression ratio is close to 8.5:1, five times the node compression ratio for the DBLP dataset of scaling factor 1 (DBLP1). Similarly, for the DBLP dataset of scaling factor 10 (DBLP10), the achieved node compression ratio is ten times the compression ratio achieved for DBLP1, close to 17:1. For DBLP10, Stripe nodes are reduced by 94% compared to the explicit storage scheme. Note that the actual number of Path Stripe nodes is constant for all three scaling factors. The compression impact on the DBLP dataset is reflected on the Path Stripe storage, as shown in Table 5.6. The size compression ratio for DBLP1 is 1.6:1, while for DBLP5 and DBLP10, the compression ratio is increased by a factor of 3.5 and 7 respectively. Regarding I/O savings, the compression storage scheme reduces the size of Path Stripes for the DBLP10 dataset by 90% compared to the size of Path Stripes required over the explicit storage scheme.

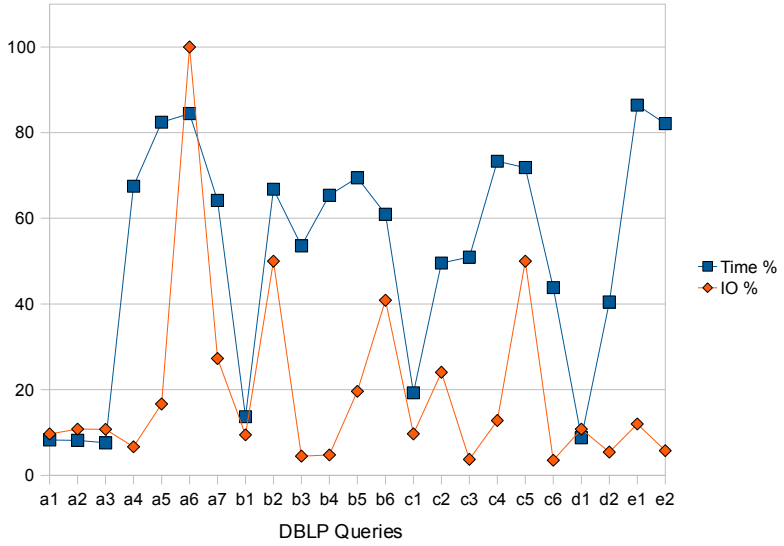
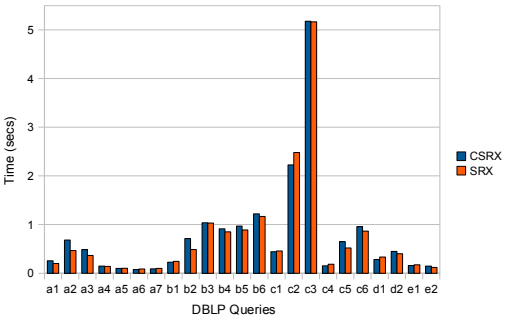


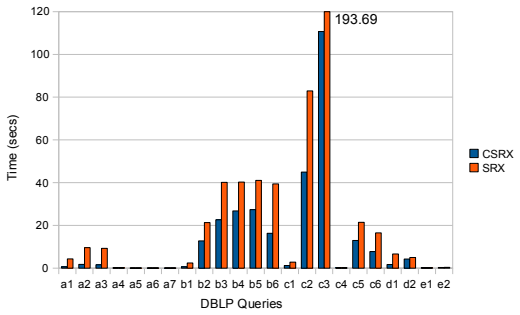
Figure 5.15: Evaluation time and I/O relationship for DBLP queries

We now present the compression effect for each of the DBLP queries, as depicted in Figure 5.14, for the largest of the DBLP datasets (sf=10, size=1GB). The results in terms of node and size reduction are impressive. Node sharing compression achieves an average node reduction of 86% which results in an average Stripe size reduction of 80% in comparison to the nodes and size required when the explicit storage scheme is used.

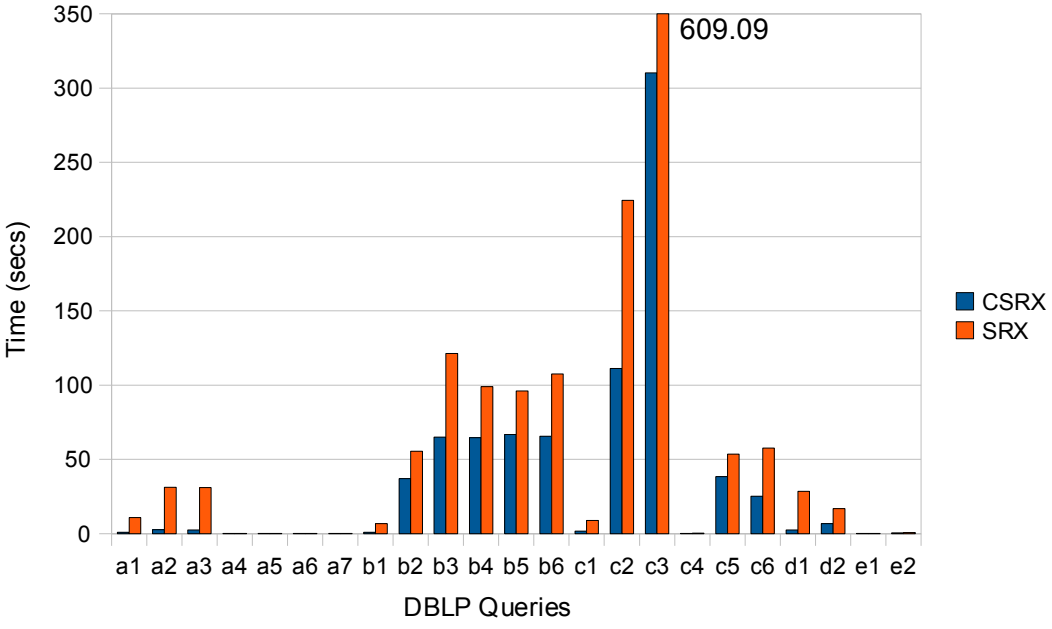
The query evaluation results for the largest of the DBLP datasets, DBLP10, are displayed in Figure 5.16(c). For all DBLP queries, CSRX is at worst comparable to SRX. On average though, CSRX evaluates DBLP queries in half the response time of SRX. Although the compression benefits are significant, one would expect an evaluation time speedup proportional to the reduction in size. As seen in Figure 5.15, where the normalised evaluation time speedup and size reduction are depicted, this does not always occur. One of the reasons is that for some of the tested queries (*e.g.,* a4-a7), the size of the required for query evaluation Stripes is relatively small even when the explicit storage scheme is used. As a result, the benefits from the reduction of the (small) I/O cost are counterbalanced by the extra computational cost for decompression and therefore the evaluation speedup is minimal. Another reason is that for some other queries (*e.g.,* b3, b4, c3), most of the query evaluation time is spent on processing value-based predicates, for which the proposed compression has no effect whatsoever. Despite that, the tree-sharing compression effect is evident on the DBLP dataset; all



(a) DBLP1



(b) DBLP5



(c) DBLP10

Figure 5.16: CSRX query evaluation for DBLP1, DBLP5 and DBLP10 datasets

queries that involve structural processing benefit by a smaller or larger factor, as the average time speedup of 48% indicates.

Regarding the compression effect on the smaller DBLP datasets, the evaluation results for scaling factors 1 (DBLP1) and 5 (DBLP5) are displayed in Figure 4.27(a) and Figure 4.27(b) respectively. For DBLP1, we observe that poor compression mostly hurts CSRX query performance rather than improving it. However, for the DBLP5 dataset, the compression ratio has significantly improved and its effect is visible; Evaluation times are improved by 40% on average compared to SRX evaluation times.

5.6.5 Comparison with MDB

We now compare our evaluation engine prototype over the tree-sharing storage scheme, CSRX, with the state-of-the-art in XML query processing, MDB. We also discuss CSRX results with a closer look at those of SRX which we regard as the baseline case for Path Stripe storage. The comparison is structured in the following manner: We first divide the testbed queries in two main categories based on the performance of CSRX against MDB. Each of the two query categories are then further divided according to the CSRX compression impact on SRX query evaluation performance. In detail, all tested queries are grouped as follows:

- Query Category C: includes all queries for which CSRX performs *better* than MDB.
  - Query Sub-Category C_1: SRX performs worse than MDB but due to the compression benefits on query evaluation, CSRX outperforms MDB.
  - Query Sub-Category C_2: SRX performs better than MDB, CSRX compression has a positive impact on query evaluation results and thus CSRX further improves evaluation performance.
  - Query Sub-Category C_3: SRX performs better than MDB and despite the negative impact of CSRX compression on query evaluation, CSRX still performs better than MDB.
- Query Category M: includes all queries for which CSRX performs *worse* than MDB.
  - Query Sub-Category M_1: SRX performs better than MDB but CSRX compression has a negative impact on query evaluation performance that results in CSRX being outperformed by MDB.
  - Query Sub-Category M_2: SRX performs worse than MDB and CSRX compression's negative effect results in further performance decrease.
  - Query Sub-Category M_3: SRX performs worse than MDB, CSRX compression has a positive impact on query evaluation results, but CSRX is still outperformed by MDB.

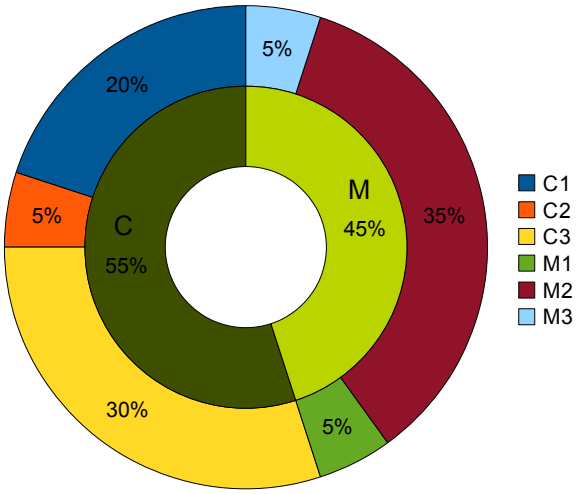


Figure 5.17: Xmark query distribution for CSRX compared to MDB (inner) and both MDB and SRX (outer) results

5.6.5.1 Xmark

The evaluation results of CSRX, SRX and MDB for the largest Xmark dataset (sf=100, size=11GB), are displayed in Figure 5.18. We also present the distribution of the Xmark queries to the query categories in Figure 5.17. Let us first summarise CSRX performance compared to SRX. As already described, CSRX tree-sharing compression has a positive impact on query evaluation, for queries a4, b4-c3 and a5, which comprise 30% of the Xmark query testbed. For those queries, CSRX evaluation results have been improved by 50%. However, for the remaining 70% of the Xmark query testbed, the low degree of compression combined with the added complexity of the evaluation algorithms, results in a performance decrease of 25%.

Regarding the comparison with MDB, CSRX outperforms MDB for the 55% of the Xmark query testbed (query category C - 11/20 queries). For those queries, CSRX improves performance by 53% on average[3]. For queries b4-c2 and e5 (sub-category C_1), CSRX compression enables our evaluation engine to outperform MDB. For query a4 (sub-category C_2), CSRX further improves SRX query performance. Finally, for queries a2, b1-b3 and d1-d2 (sub-category C_3), the CSRX compression results in a query performance decrease. Nevertheless, CSRX still performs better than MDB.

On the other hand, for 45% of the Xmark testbed (query category M - 9/20 queries), MDB performs better than CSRX by 69% on average. For query a3 (sub-category M_1),

[3]Excluding the result for query d2, that MDB was not able to evaluate.

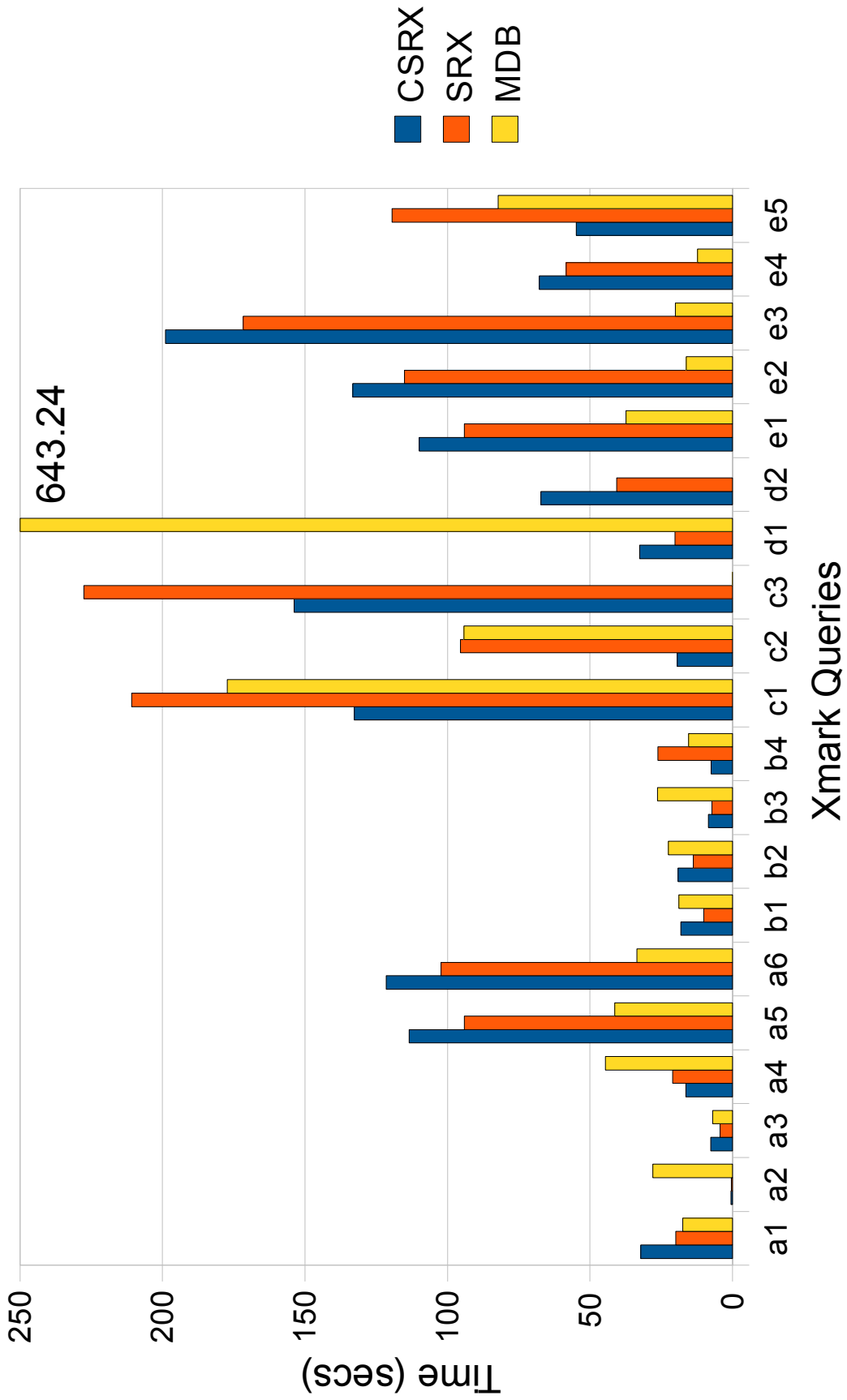


Figure 5.18: CSRX and MDB comparison for the Xmark100 dataset

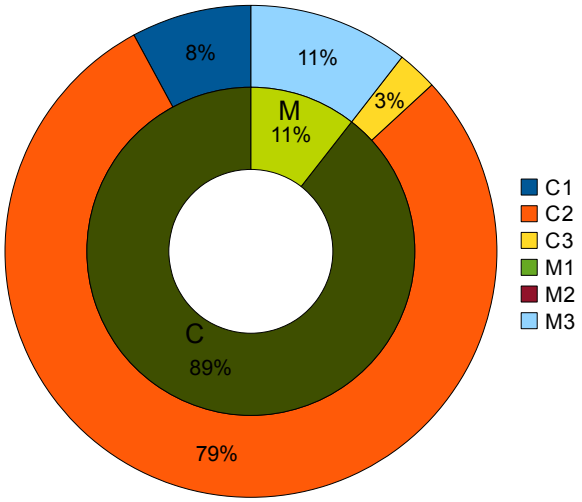


Figure 5.19: Mbench query distribution for CSRX compared to MDB (inner) and both MDB and SRX (outer) results

although SRX performs better than MDB, the CSRX compression results in poorer (compared to MDB) performance. The exact opposite occurs for query c3 (sub-category M_3), where despite the performance increase by a large factor due to CSRX compression, MDB is still the clear winner due to the very selective value-based predicates. Finally, there exists a large query sub-category, M_2, consisting of queries a1, a5-a6 and e1-e4, where the CSRX compression, results in query performance decrease, while SRX is already outperformed by MDB.

5.6.5.2 Mbench

We now proceed to the evaluation results for the largest Mbench dataset (sf=10, size=5GB), as shown in Figure 5.20. We also provide the Mbench query distribution in Figure 5.19. As already described, CSRX compression has a huge impact on query evaluation; for all Mbench queries but one (h2), CSRX outperforms SRX by a large factor (37%).

Regarding the comparison with MDB, we begin with query category M. MDB outperforms CSRX for 11% of the Mbench query testbed (4/38 Mbench queries). For these queries, namely queries b1-b3 and d3, MDB is 23% faster than CSRX on average, due to very selective value-based predicates, as already described in Section 4.5.4. Note, that these queries belong to sub-category M_3, which means that CSRX performs better than SRX although it is still outperformed by MDB for the reasons mentioned above.

The majority of Mbench queries, however, fall into query category C; for 89% of

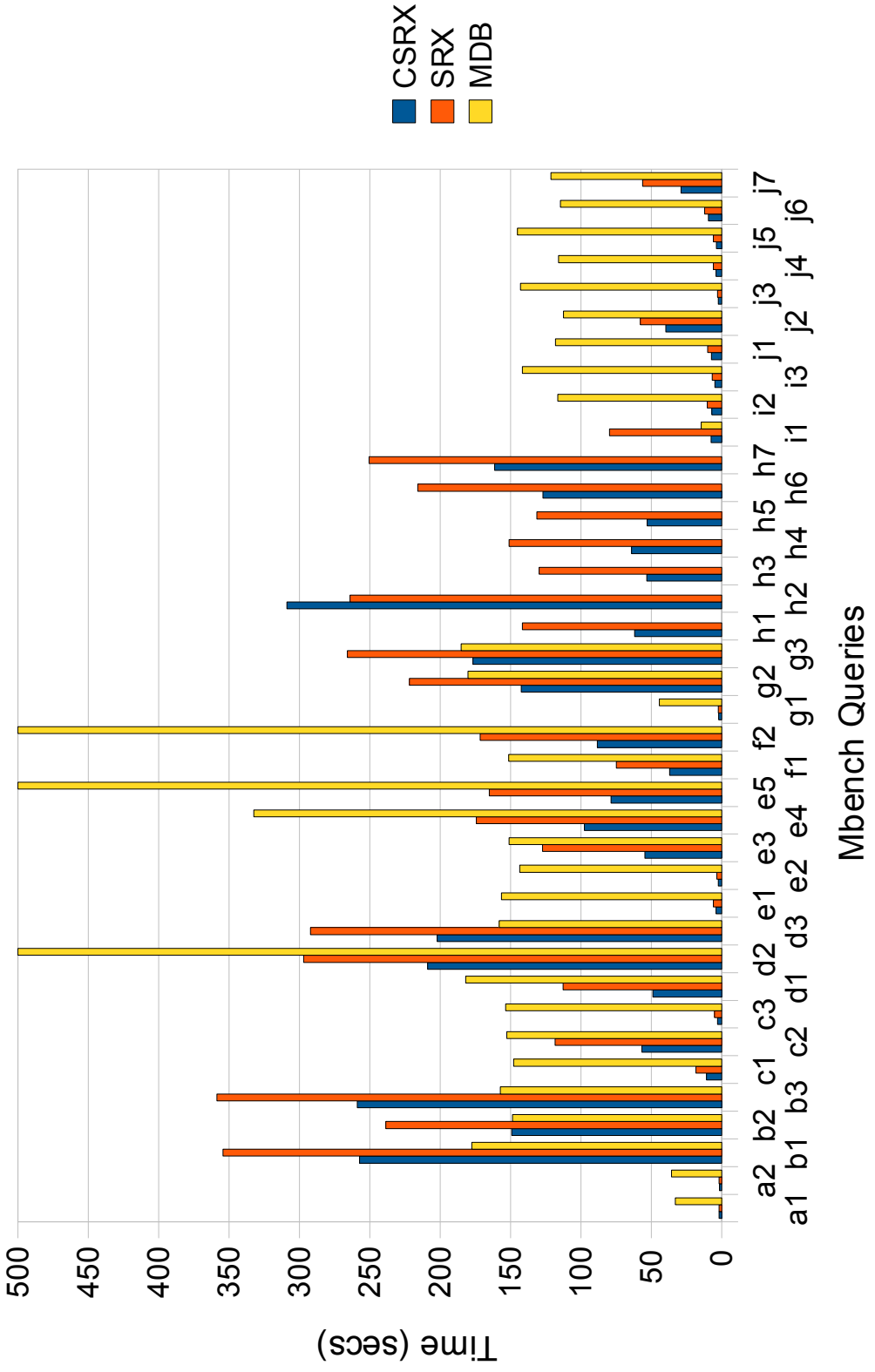


Figure 5.20: CSRX and MDB comparison for the Mbench10 dataset

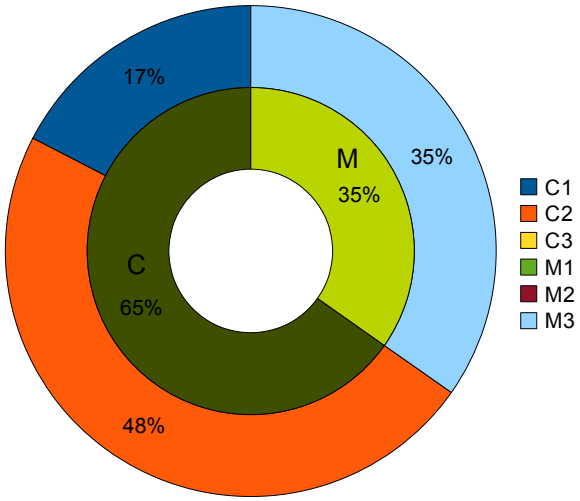


Figure 5.21: DBLP query distribution for CSRX compared to MDB (inner) and both MDB and SRX (outer) results

the Mbench query testbed (34/38 Mbench queries), CSRX outperforms MDB, achieving a performance improvement of 80% on average[4]. It is also evident that the largest query sub-category, is the one that SRX already outperforms MDB (sub-category C_2); for 79% of the Mbench queries, CSRX compression further improves SRX performance. There is also a small number of queries, g2-g3 and i1 (sub-category C_1), for which although SRX performs worse than MDB, CSRX compression results in CSRX outperforming MDB. Finally, there is a single Mbench query, h2, that CSRX performs worse than SRX; MDB, however, fails to evaluate this query (as well as queries h1-h7).

5.6.5.3 DBLP

Finally, we present the evaluation results for the largest DBLP dataset (sf=10, size=1GB), as shown in Figure 5.22. We have already described that CSRX performs better than SRX for all DBLP queries by 48% on average.

CSRX outperforms MDB for 65% of the DBLP query testbed (15/23 DBLP queries) by 88% on average. These queries are distributed to sub-categories as follows: Queries a1-a3 and c6 belong to sub-category C_1, where the added effect of CSRX compression, accelerates query evaluation so that it outperforms MDB. The largest sub-category however, as in the case of the Mbench dataset, is sub-category C_2. For queries a4-b1,

[4]Excluding the results for queries h1-h7, that MDB was not able to evaluate.

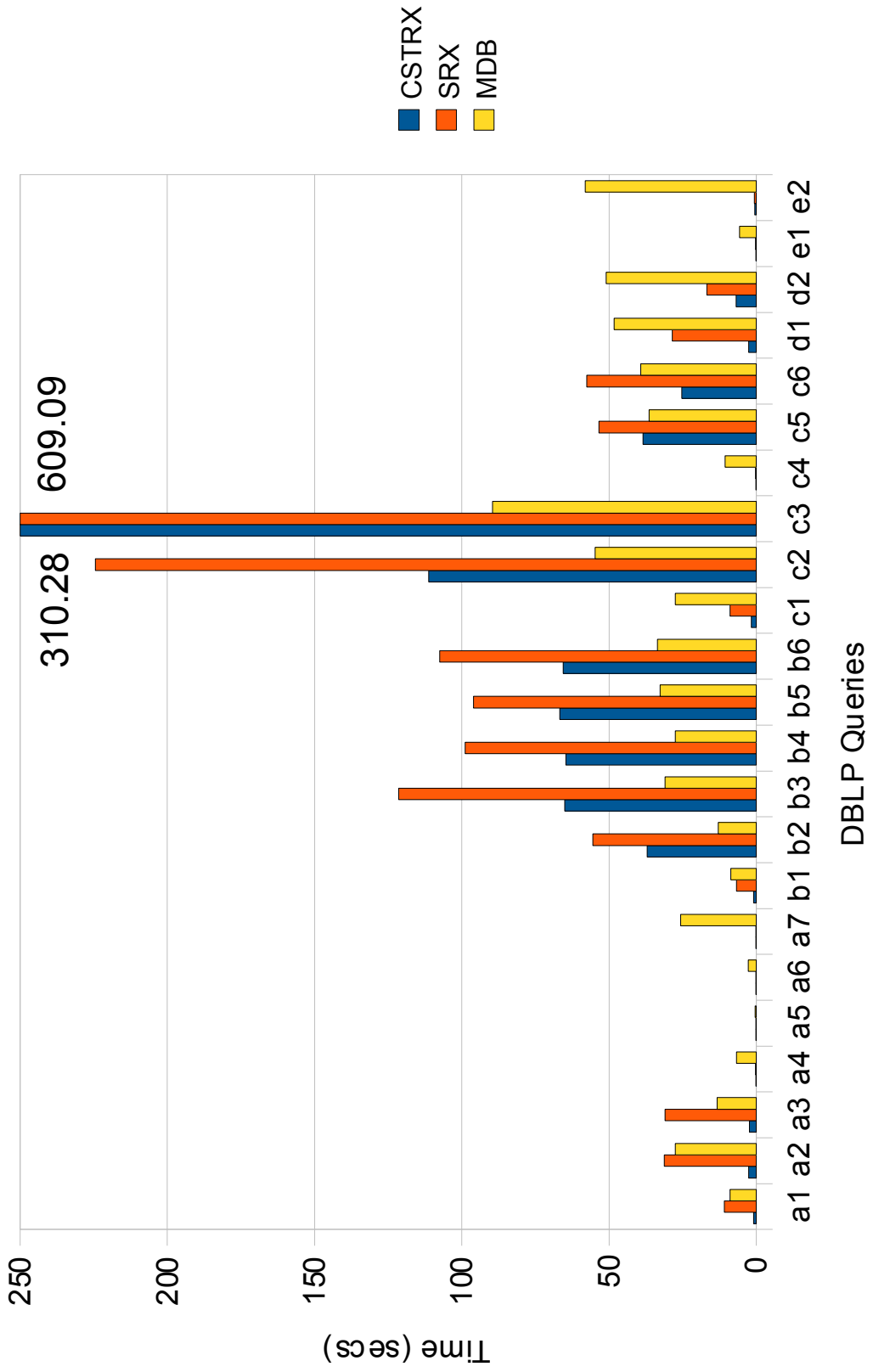


Figure 5.22: CSRX and MDB comparison for the DBLP10 dataset

c1, c4 and d1-e2, the added effect of CSRX compression, results in further improvement of SRX query evaluation performance.

On the other hand, CSRX underperforms for 35% of the DBLP query testbed (query category *M* - 15/23 queries), by 50% on average compared to MDB. As seen in Figure 5.21, query category *M* coincides with sub-category M_3; for all such DBLP queries, namely queries b2-b6, c2-c3 and c5, CSRX achieves an important query performance increase with respect to SRX, due to the CSRX compression effect. However, MDB still outperforms CSRX as these queries contain very selective value-based predicates that dominate the overall query evaluation performance.

Chapter 6

Structural Agnostic Compression Storage Scheme

In the previous chapter, we extended the basic Striping storage model with respect to the structural regularities that an XML document may present. We compressed the structural part of the document in a more compact representation to reduce its storage cost and, thus, the I/O cost during query evaluation. This representation, as shown, is closely dependent on the original XML tree structure and therefore its effect can significantly vary. So, the question that naturally arises is whether the Striping storage model can be extended in a structure-agnostic way so that it results in a more concise representation compared to the original representation, yet being immune to the structural regularities of the XML dataset.

The rest of the chapter is organised as follows: We begin by presenting the structural agnostic compression scheme (Section 6.1), and the three alternative compression methods which are considered for structural agnostic compression (Section 6.2). We also discuss Stripe storage issues (Section 6.3). We continue with the document loading process to our native store over the proposed storage scheme (Section 6.4), and the tree node reconstruction from compressed Stripe nodes (Section 6.5). Lastly, we conclude with the experimental results for the proposed storage scheme (Section 6.6) and the related work regarding XML compression (Section 6.7).

6.1 Structural Compression

As described in Section 2.2, the structure of the XML dataset is partitioned in a set of Path Stripes. We create a Path Stripe for each unique label-path p from the document

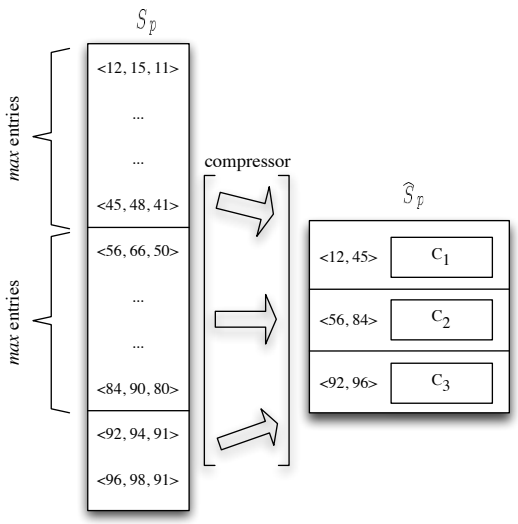


Figure 6.1: Path Stripe agnostic compression example

root node; all XML nodes $n \in p$ are then stored at the equivalent Stripe S_p. Each XML node n is stored as an $\langle start, end, par \rangle$ triplet.

The idea for structural agnostic compression is to compress each Path Stripe (as defined in the explicit storage model) as follows: Let S_p be a Path Stripe constructed for label-path p containing k XML element nodes *i.e.,* $\{n_1, \ldots, n_k\}$. We define a parameter *max* and partition S_p in m node chunks $C_1, \ldots, C_m$ so that each chunk contains at most *max* number of nodes and thus $m = \lfloor \frac{k}{max} \rfloor$. Each node chunk C_i is then independently compressed to form a compressed node chunk $\hat{C}_i$ with $i \in [1, m]$. The compressed node chunk $\hat{C}_i$, is stored as a compressed node N_i along with the *start* value of the boundary nodes of the corresponding chunk. The sequence of all compressed nodes N_i that correspond to the chunks of the original Path Stripe S_p, form the new, compressed Path Stripe $\hat{S}_p$. To summarise, a Path Stripe node is comprised of: (a) min_start, the *start* position of the first element node (b) max_start, the *start* position of the last element node and (c) $\hat{C}$, the compressed node chunk that contains all element nodes between min_start and max_start.

Example 6.1: Consider the Path Stripe S_p for some label-path p in Figure 6.1. The first *max* number of nodes, *i.e.,* from node (12,15) to node (45,48), define node chunk C_1. Chunk C_1 is compressed to $\hat{C}_1$ and stored to the Path Stripe node N_1, along with the minimum and maximum *start* values of their corresponding original nodes. Thus, we have that $N_1 : \langle 12, 45, \hat{C}_1 \rangle$. Likewise, the next *max* number of nodes define node chuck

C_2 which is compressed in $\hat{C_2}$ and stored to Stripe node $N_2 : \langle 56, 84, \hat{C_2} \rangle$. Finally, the remaining nodes are compressed in Stripe node $N_3 : \langle 92, 96, \hat{C_3} \rangle$. □

6.2 Compression Methods

A node chunk of k nodes is an XML element node sequence of size k. The serialised representation of an element node is that of the $\langle start, end, par \rangle$ triplet as used in the explicit storage scheme for serialising element nodes in Path Stripes. Thus, we have that the node chunk is essentially, a block of numerical data. A node chunk $\langle n_1, n_2, \ldots, n_k \rangle$ is $\langle \langle start_1, end_1, par_1 \rangle, \langle start_2, end_2, par_2 \rangle, \ldots, \langle start_k, end_k, par_k \rangle \rangle$.

We have considered the following three options for compressing a node chunk: (a) the Dewey encoding, (b) the lossless data compression algorithm, *bzip2*, available in the *bzip2* library, and (c) the lossless data compression algorithm, *Deflate*, available in the *zlib* library.

6.2.1 Dewey Encoding

The Dewey Decimal Classification [69] is a system for library classification, developed for general knowledge organisation. In the context of XML, it was first appeared in an attempt to provide a code, the *Dewey Order*, that effectively captures XML node ordering when XML is stored in relational databases [95]. The Dewey Order encoding for an XML node n is a vector which represents the path from the document root to n. Dewey encoding was quickly adopted as an hierarchical, prefix-based labelling scheme for encoding XML nodes. In [82], a variation of the Dewey order is presented, *ORDPATHs*, to accommodate node insertions without the need of re-labelling parts of the document tree. Prefix-based codes also present optimisation opportunities in the area of XML query processing. One of the merits of a Dewey code is that it provides the ancestor information of a node without the need of actually accessing those. This property is exploited in [66] for efficient processing of twig pattern matchings.

Due to the large acceptance of prefix-based encodings, researchers have studied succinct representations of Dewey codes in an attempt to minimise its size and thus storage requirements. In [95], a Dewey code is encoded using UTF-8 characters for each of the ordinals of the Dewey code. In [82, 52], Dewey codes are encoded as variable length bit-strings providing a compressed Dewey format. Each ordinal value is encoded as a variable length L_i/O_i pair. Variable length bit-string O_i encodes the

ordinal value (in fact a relative value), while its proceeding variable length bit-string L_i encodes the length of O_i. The concatenation of all L_i/O_i pairs produces the final compressed Dewey code.

6.2.2 Bzip2 Compression

Bzip2 [89] is an open source tool for text compression. The compressor, along with its associated library libbzip2, use a lossless data compression algorithm that is based on a combination of Burrows-Wheeler transform and Huffman coding.

The Burrows-Wheeler transform [21] permutes a block of data by applying a sorting algorithm such that character sequences that occur frequently are transformed into strings of identical letters and therefore can be efficiently compressed. The resulting output block contains exactly the same characters as the original data block and the only difference is that data is now placed in a different order. The Burrows-Wheeler transform is lossless and thus, completely reversible. On top of the transform, Huffman coding [54] is used for replacing fixed length symbols with variable length codes based on the frequency of use; commonly-used symbols are replaced with shorter codes while less commonly used symbols are replaced with longer codes.

6.2.3 Zlib Compression

Like the bzip2 compressor, *gzip* [42] is an open source tool for text compression. The compression tool, along with the related compression library *zlib* [43], use the *Deflate* lossless compression algorithm [34] that is based on a combination of the Lempel-Ziv algorithm and Huffman coding.

The Lempel-Ziv algorithm (LZ77) [107] , compresses data by replacing character sequences with references to the matching sequences that have already been processed. A matching character sequence is encoded by a *length/offset* pair, denoting the that next *length* characters are the same as the sequence occurred at *offset* position relative to the first character of the match. On top of the LZ77 algorithm, Huffman coding [54] is used for replacing character sequences as well as *length/offset* pairs, with variable length codes based on the frequency of use. The process is the same as described for the bzip2 compression algorithm.

It has been reported [1] that bzip2 compresses most files more effectively that the deflate compression algorithm, creating 15% smaller files than deflate. However, it

is considerably slower both during compression and most importantly during decompression by four to twelve times in comparison to zlib (deflate/inflate algorithms).

6.2.4 Node Chunk Compression

As already described, the serialised chunk of k nodes is a sequence of numerical data: $start_1.end_1.par_1.start_2.end_2.par_2 \ldots start_k.end_k.par_k$. We compress such a data block by applying either of the three compression methods described above.

As far as the Dewey encoding is concerned, it is defined so that it compresses better small values compared to large values. A large ordinal value results in a long O_i bit-string and thus in large L_i bit-string value, the length of the O_i bit-string. As a result, the compression ratio of the compressed node chunks is expected to degrade as more element nodes are compressed in Path Stripes. In addition, the average compression ratio is closely coupled with the document size; this is definitely an undesired effect. Similarly, the deflate compression algorithm is also expected to compress poorly as *start*, *end* and *par* values are constantly increasing. However, it is not expected to be affected at the same level as the Dewey encoding due to the matching sequences identification by the Lempel-Ziv algorithm. On the other hand, the bzip2 compression algorithm is expected to be less affected due to the Burrows-Wheeler transform. In any case though, constantly increasing values of *start*, *end* and *par* values do not present good compression guarantees.

To increase the achieved compression of all compression methods considered, we employ a delta encoding transform to the original node chunk data block; instead of compressing the absolute *start*, *end* and *par* values of each node, we merely replace them by delta values (offsets) of the previous or current node values in the node chuck. Each node's *start* absolute value ($n_i.start$) is replaced by the offset of its *start* absolute value compared to the absolute *end* value of its proceeding node (in the node chunk), $n_{i-1}.end$. Then, its *end* absolute value is replaced by the offset of its *end* absolute value compared to its *start* absolute value. Finally, its *par* absolute value is replaced by the offset of its *start* absolute value compared to its *par* absolute value. The pseudocode for delta transform is shown in Figure 6.2.

The application of encoding delta values instead of absolute values has many desired effects. First of all, it replaces large ordinal values with smaller ones, which will result in better compression ratios when the Dewey encoding is used. Apart from that, the other two compression methods benefit from delta encoding, since the delta trans-

Algorithm 6.1: applyDeltaEncoding

Data: node chuck: $\langle n_1, \ldots n_k \rangle$

Result: Applies delta encoding on node chunk $\langle n_1, \ldots n_k \rangle$

```
1 begin
2     acc ← n_i.start ;
3     foreach n_i ∈ ⟨n_1, ..., n_k⟩ do
4         n_i.start ← n_i.start − acc ;
5         acc ← acc + n_i.start ;   // becomes the absolute value n_i.start
6         n_i.par ← acc − n_i.par ;
7         n_i.end ← n_i.end − acc ;
8         acc ← acc + n_i.end ;      // becomes the absolute value n_i.end
9 end
```

Figure 6.2: Applying delta encoding on a node chunk

form increases the probability of the delta values to occur more frequently in the node chunk. This can be exploited firstly from the LZ77 algorithm that will replace the same occurrences by *length/offset* pairs but also from Huffman coding that both the deflate and bzip2 algorithms are using, resulting in better compression ratios.

6.3 Stripe Storage

As described in Section 6.1, a Path Stripe node is made of three attributes: (a) *min_start*, the *start* position of the first tree node being shared, (b) *max_start*, the *start* position of the last tree node being shared and (c) $\hat{C}$, the compressed node chunk that contains all tree nodes between *min_start* and *max_start*.

All Path Stripes are stored in B$^+$-tree structures with *max_start* value acting as the B$^+$-tree key. Each Path Stripe node stores compressed XML element nodes in ascending *start* order. Note that the definition of compressed Path Stripes, enforces that each Stripe node contains continuous element nodes (of a certain path) in document order. This means that for any two Stripe nodes N_i, N_{i+1}, we have that:

$$N_i.min\_start < N_i.max\_start < N_{i+1}.min\_start < N_{i+1}.max\_start$$

which effectively imposes a relative order of the element nodes that are contained in Stripe nodes; an element node $n \in N_i$ always occurs *before* any of the elements

$n' \in N_{i+1}$. This observation, along with the fact that all Path Stripe nodes are sorted on *max_start* value, enforces that a Path Stripe stores all contained XML elements in document order.

6.4 Document Loading

We now describe the loading process of an XML document into the compressed XML store. The loading of Attribute and Value Stripes is performed as described in Section 4.3 and is therefore omitted from this section.

As far as the document structure loading is concerned, the process resembles the loading process used for the explicit storage scheme. Likewise, we build a main memory tree, the skeleton, reflecting all unique paths of the document tree. When the end of an element node is encountered, we identify the structural information of that node and insert it into the appropriate Path Stripe. Each skeleton node corresponds to a unique document path and thus to a unique Path Stripe structure.

The basic difference lies in the insertion of the element nodes to the Path Stripe. Now, each skeleton node is equipped with a compressor that builds node chunks of *max* number of element nodes. As soon as a node is consumed, instead of adding it directly to the Path Stripe as in the case of the explicit storage scheme, it is now added to an active node chunk. When the *max* number of nodes is reached, the active node chunk is compressed and then added to the appropriate Path Stripe, along with the *start* value of the chunk boundary nodes. The process continues in the same manner until the last element of the document is processed. The loading process then finishes by compressing the remaining nodes of each skeleton node still having an active node chuck, and their insertion to the corresponding Path Stripes.

Determining the *max* number of element nodes that are compressed in a node chunk and thus in a single Path Stripe node is not trivial as there are many parameters that must be taken into account. Maximising the *max* parameter, *i.e.,* the number of nodes in a chunk, will result in better compression, as the probability of having repeated occurrences increases. However, restrictions imposed by the access method used for Stripe storage and the underlying storage manager in general must also taken into account. For instance, a compressed Path Stripe node must not exceed the physical page size (typically 4K), as this would cause the node entry to be placed to overflow pages causing significant performance degrade. In fact, the compressed Path Stripe node is restricted to be much smaller than page size. As already described we use

B$^+$-tree structures for Stripe storage. The B$^+$-tree implementation we use for Stripe storage uses the following formula to define the maximum size of a B$^+$-tree key or payload that can be accommodated in a leaf page:

$$maximum\_size = page\_size/(minimum\_keys * 2)$$

where *minimum_keys* is the minimum number of keys stored on each page and this value is multiplied by two because each key/data pair requires two slots on a B$^+$-tree page [3]. Thus, a typical page size of 4KB and minimum acceptable value of two keys per page, effectively restricts the maximum size of a Path Stripe node and thus of a compressed node chunk close to 1KB. Therefore, our goal is to identify the *max* number of nodes that their compressed size (*i.e.,* of the compressed node chunk) will not exceed 1KB.

When Dewey encoding is used to compress node chunks, it is easy to keep track of the compressed size of the node chunk; every time a node element is serialised, we apply the delta encoding and then the bit-strings for delta values of *start*, *end* and *par* values are appended to the Dewey bit stream. As soon as the size of the bit stream approaches the permitted maximum size, we insert the Path node in the corresponding Stripe.

However, for applying the bzip2 compression algorithm the node chunk must be available in priori. To that end, we employ the following strategy: we begin with a fixed size of node chunk. As soon as enough nodes are serialised to reach the node chunk size, we compress it. If its compressed size is less than the maximum size permitted for storage at a B$^+$-tree page, we simply insert the Path node to the Stripe. If this does not hold, we restrict the size of the node chunk and try compressing it again. The process continues until the restricted chunk is compressed to fit in a page. The nodes of the chunk that were not compressed, remain in the chunk so that they will be compressed later. Then the size of the node chunk that will be compressed next time is adjusted based on the size of the node chuck that was actually compressed and the average compression ratio obtained so far. This will gradually calculate the node chunk size that, when compressed, satisfies the storage restrictions. We also apply the same strategy when using the deflate compression algorithm for node chunk compression. Note that this is not imposed by the algorithm, as it permits a gradual compression like in the Dewey encoding. Nevertheless, we chose to use the same strategy so we can have a direct comparison between the two compression algorithms, bzip2 and deflate.

6.5 Node Reconstruction

We now describe the inverse process of retrieving a Stripe Node and reconstructing XML nodes to their serialised form: $\langle start, end, par, level, kind, label, text \rangle$. As far as the Attribute and Value Stripes are concerned, the process is the same as described in Section 4.4.

Path Stripes contain compressed chunks of tree element nodes. As soon as a Path Stripe node is retrieved, the node reconstruction process requires the following actions:

Node Chunk Decompression The compressed node chunk is decompressed to the original node chunk data block.

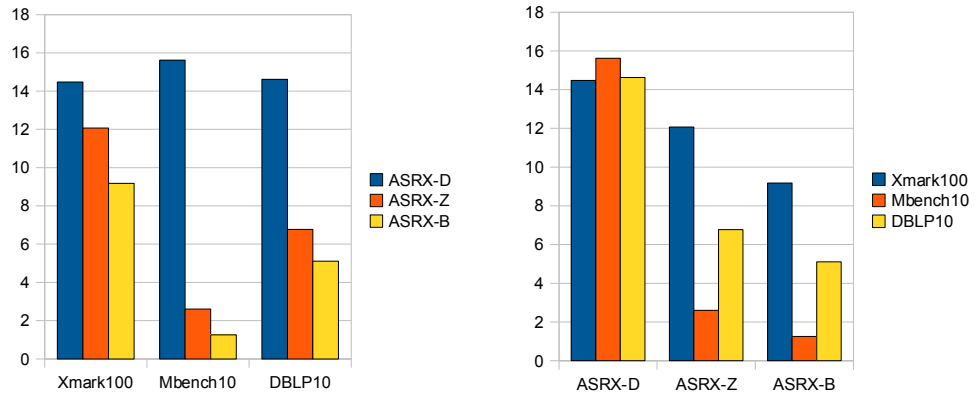
Inverse Delta Transform The inverse of the delta transform is applied to the data block of the decompressed node chunk so that all relative node values are transformed back to the absolute node position values.

Node Reconstruction Since we have reconstructed the *start*, *end* and *par* values for all tree nodes in the chunk, each of them can now be returned enriched with *level*, *label* and *kind* information derived from the Stripe, as described in Section 4.4.

6.6 Experimental Results

We now present our experimental results regarding the proposed agnostic compression storage scheme. Our experimental setup remains the same as described in Section 4.5. The query engine of our native store prototype is largely unaffected by the underlying storage scheme; we merely needed to change the Scan operators, to accommodate compressed Path Stripe node retrieval. All evaluation algorithms, in contrast to the tree-sharing compression storage scheme, are exactly the same as the ones used by our query engine over the explicit storage scheme.

Our experimental study is, as usual, divided in three parts, for the Xmark, Mbench and DBLP datasets. We shredded all XML documents in our native store using all three variants of agnostic compression storage scheme. To demonstrate the benefits and/or drawbacks of agnostic compression, we directly compare our results to those of the explicit (uncompressed) storage scheme, SRX. We also compare the three proposed compression methods. To disambiguate our query engines, our native XML store over the agnostic compression storage scheme is hereafter noted as ASRX. In addition, to designate the compression method, we use the shorthands ASRX-D, ASRX-Z and



(a) grouped by dataset (b) grouped by compression method

Figure 6.3: Overview of agnostic compression effectiveness on Path Stripe storage

ASRX-B for Dewey encoding, zlib and bzip2 compression respectively. Finally, during query evaluation, we have enabled all possible optimisations and thus all query engines operate using the PRO evaluation setup.

To measure the compression effectiveness of the proposed compression methods, we use the compression ratios defined in Section 5.6.1. The node compression ratio is now defined as: $CR_n = \frac{N_{SRX}}{N_{ASRX}}$, while size compression ratio is: $CR_p = \frac{P_{SRX}}{P_{ASRX}}$.

We now provide an overview of the compression effectiveness for all three datasets: Xmark, Mbench and DBLP. In Figure 6.3, we present the normalised Path Stripe size reduction achieved for the largest of each of the XML datasets, against the Path Stripe size when no compression occurs. As seen from the graph, agnostic compression has a huge impact on the size of the compressed Path Stripes for all datasets. The Path Stripe size is reduced by at least 85% regardless of the compression method used or the dataset characteristics. Regarding the three proposed compression methods, we notice that when the Dewey encoding is used, the size reduction is similar for all datasets. However, when the zlib and bzip2 compression methods are used, the compression impact significantly varies. For the DBLP dataset, both ASRX-Z and ASRX-B further reduce the Path Stripe size of ASRX-D by a factor of 2. Finally, for the Mbench dataset, ASRX-Z and ASRX-B further reduce Path Stripe size by a factor of 5 and 7 respectively.

Xmark Scaling Factor	Path Stripe Cardinality (Nodes)				CR_n		
	SRX	ASRX-D	ASRX-Z	ASRX-B	ASRX-D	ASRX-Z	ASRX-B
0.1	167866	799	798	748	210.1	210.36	224.42
1	1666316	4530	4011	3230	367.84	415.44	515.89
10	16703211	42985	36389	27738	388.58	459.02	602.18
100	167095845	430760	359082	272770	387.91	465.34	612.59

Table 6.1: Compression effect on Path Stripe cardinality for Xmark datasets

Xmark Scaling Factor	Path Stripe Size (Pages)				CR_p		
	SRX	ASRX-D	ASRX-Z	ASRX-B	ASRX-D	ASRX-Z	ASRX-B
0.1	1526	558	525	515	2.73	2.91	2.96
1	10600	1865	1644	1366	5.68	6.45	7.76
10	100625	14793	12545	9629	6.8	8.02	10.45
100	999855	144724	120704	91769	6.91	8.28	10.9

Table 6.2: Compression effect on Path Stripe size for Xmark datasets

6.6.1 Xmark

We now take a closer look at the impact of the structural agnostic compression on the Xmark dataset. In Table 6.1, we provide details regarding the cardinality of Path Stripes when the explicit and agnostic compression storage schemes are used. We also compare the compression effectiveness of all three compression methods described in Section 6.2. We first observe that regardless of the compression method, the achieved node compression ratio CR_n for the smallest Xmark dataset (sf=0.1) is close or less than half of the node ratios achieved for all other Xmark datasets. This is due to the low Stripe cardinalities resulting from the dataset's small size. For most Stripes, the total number of nodes (in SRX) is much smaller than *max*, the maximum number of nodes that fit in a node chunk. Observe that the total number of the compressed Stripe nodes for Xmark0.1 is at most 800 and these are distributed to 515 Path Stripes. For the rest of the Xmark datasets, each of the compression methods achieves high node compression ratios; ASRX-D, achieves the lowest node compression ratios, compared to the two other compression methods; node compression ratio for the Xmark1 dataset, ranges between 367:1 and 388:1. ASRX-Z always performs better than ASRX-D, achieving node compression ratios between 415:1 and 465:1. Similarly, ASRX-B always performs better than ASRX-Z (and thus ASRX-D), with node compression ratios ranging from 515:1 to 612:1. To emphasise the compression effectiveness of the

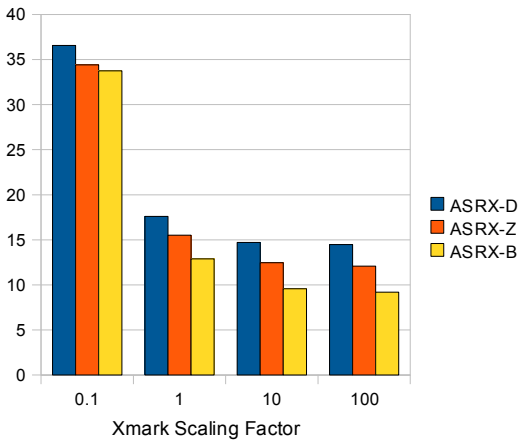


Figure 6.4: Path Stripe size compression for Xmark datasets

agnostic compression storage scheme, we report that for any of the Xmark datasets and regardless of the compression method used, the average cardinality of the compressed Stripes is less than 0.5% of the Path Stripe cardinality of the explicit storage scheme.

In addition to Path Stripe cardinality, in Table 6.2, we report Path Stripe storage information for the agnostic compression storage scheme. We observe that the trends for size compression ratios follow the node compression ratios: For all Xmark datasets, ASRX-B always achieves the best size compression ratio CR_p, reaching up to 11:1 for Xmark100. On the contrary, ASRX-D always produces the worst size compression ratios (7:1 for Xmark100). Finally, ASRX-Z achieves compression ratios that are between the other two compression methods (8.3:1 for Xmark100); however its compression effect is closer to ASRX-D than ASRX-B. All three compression methods have a huge impact on Path Stripe storage. In Figure 6.4, the size of the compressed Path Stripes for each of the proposed compression methods is depicted. The results are normalised with respect to the Path Stripe size of the explicit storage scheme. We verify that:

- Agnostic compression has a big impact on Path Stripe storage for the Xmark dataset. Regardless of the compression method used, the size of Path Stripes is reduced by *at least* 64%.

- For the smallest Xmark dataset, the achieved compression, although significant, is much worse than the compression achieved for the rest of the Xmark datasets. This occurs due to the small number of Path nodes per Stripe, for the majority of Path Stripes.

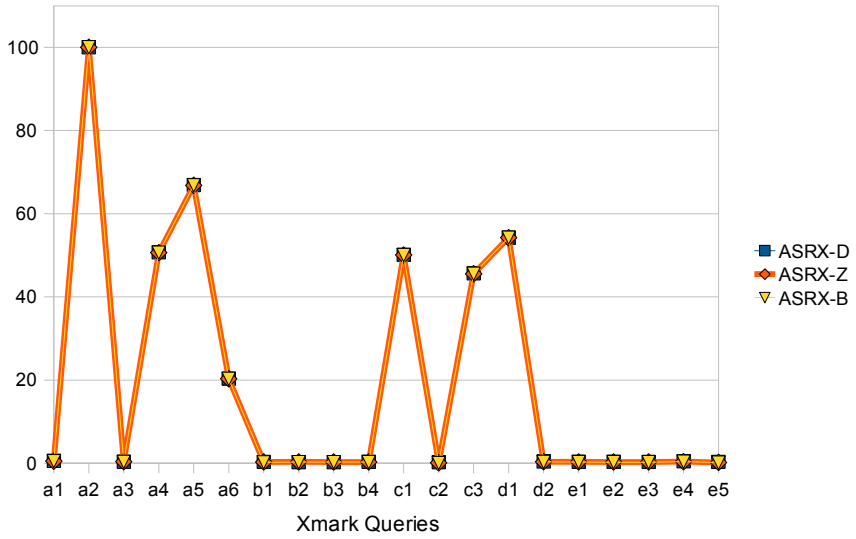
- Agnostic compression using the Dewey encoding (ASRX-D), reduces the original Path Stripe size (explicit scheme) by 82% for Xmark1 and 85% for Xmark10 and Xmark100 datasets.

- Agnostic compression using the Zlib compression algorithm (ASRX-Z), always produces better compression ratios for the Xmark dataset, compared to the Dewey encoding. It further reduces total Path Stripe size by 2%.

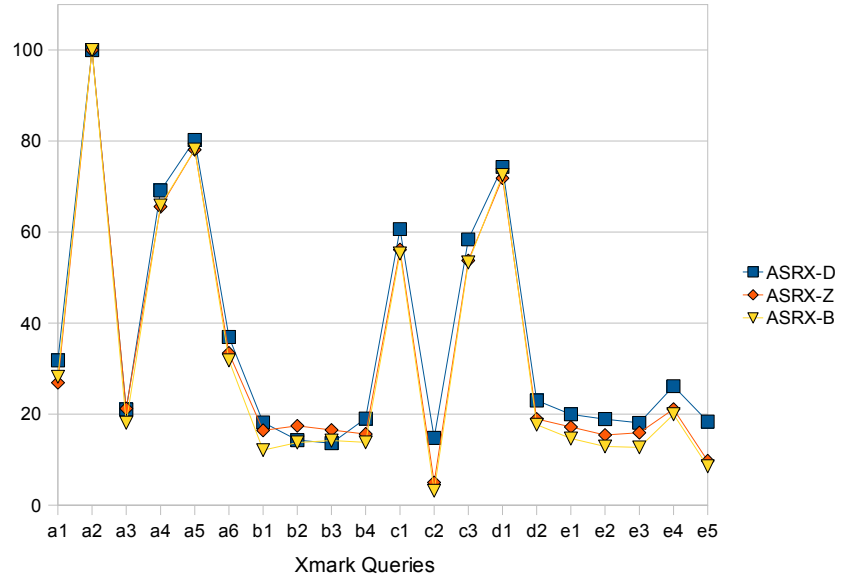
- Agnostic compression using the Bzip compression algorithm (ASRX-B), always produces the best compression ratios for the Xmark dataset. Compared to ASRX-Z, it further reduces total Path Stripe size by 3%.

We now regard the compression effect in terms of node and size reduction of the Stripes that are considered for each of the Xmark queries in isolation. The results for the largest of the Xmark datasets (sf=100, size=11GB) are depicted in Figure 6.5. As seen in Figure 6.5(a), for more than half of the Xmark queries, the required Stripes that need to be accessed, contain less than 1% of the Stripe nodes that are stored using the explicit storage scheme. Even for the rest of the queries though, the accomplished reduction due to compression is significant, ranging from 30% to 80% (query a2 is an exception as no Path Stripes are selected). The average node reduction achieved by agnostic compression is close to 80%, regardless of the compression method used. The size reduction per Xmark query is proportional to the node reduction, as seen in Figure 6.5(b). The average size reduction ranges from 63% to 68%, depending on the compression method used. Overall, the compression effect is evident for Xmark queries and is expected to have an impact on query evaluation.

We now compare the three compression methods proposed for the agnostic compression storage scheme against the explicit storage scheme, in terms of query evaluation performance. In Figure 6.6, we present the actual evaluation times for the largest of the Xmark datasets (sf=100, size=11GB). As seen from the graph, our query engine over the agnostic compression storage scheme outperforms by a large factor the one over our explicit storage scheme. Even when the Dewey encoding is used for Path Stripe compression, which performs worse than the other two alternative compression methods, the benefits of compression on query evaluation performance are evident. Compared to SRX, ASRX-D achieves a query evaluation speedup of 68.9% on average for Xmark queries. However, as already mentioned and for the majority of Xmark queries, it performs worse than ASRX-Z and ASRX-B; an expected result considering



(a) Node reduction



(b) Size reduction

Figure 6.5: Agnostic compression effect for Xmark queries

that it produces lower compression ratios. The cases where ASRX-D is comparable to the two other alternatives are the cases where its compression impact is similar to (or slightly better than) ASRX-Z and ASRX-B (*e.g.,* a4, b1-b3). Regarding ASRX-Z and ASRX-B, there does not seem to be a clear winner, at least not for the Xmark dataset. Each of them performs better than the other for about half of the tested queries. ASRX-Z achieves an average speedup of 73.4% for Xmark queries, while ASRX-B's achieved

average speedup is 73.7%. For those queries that the compression of ASRX-Z is better or similar to the compression achieved by ASRX-B, ASRX-Z is faster due to that it requires less time for decompression compared to ASRX-B. On the other hand, ASRX-B is faster for those queries that it achieves a better compression ratio than ASRX-Z.

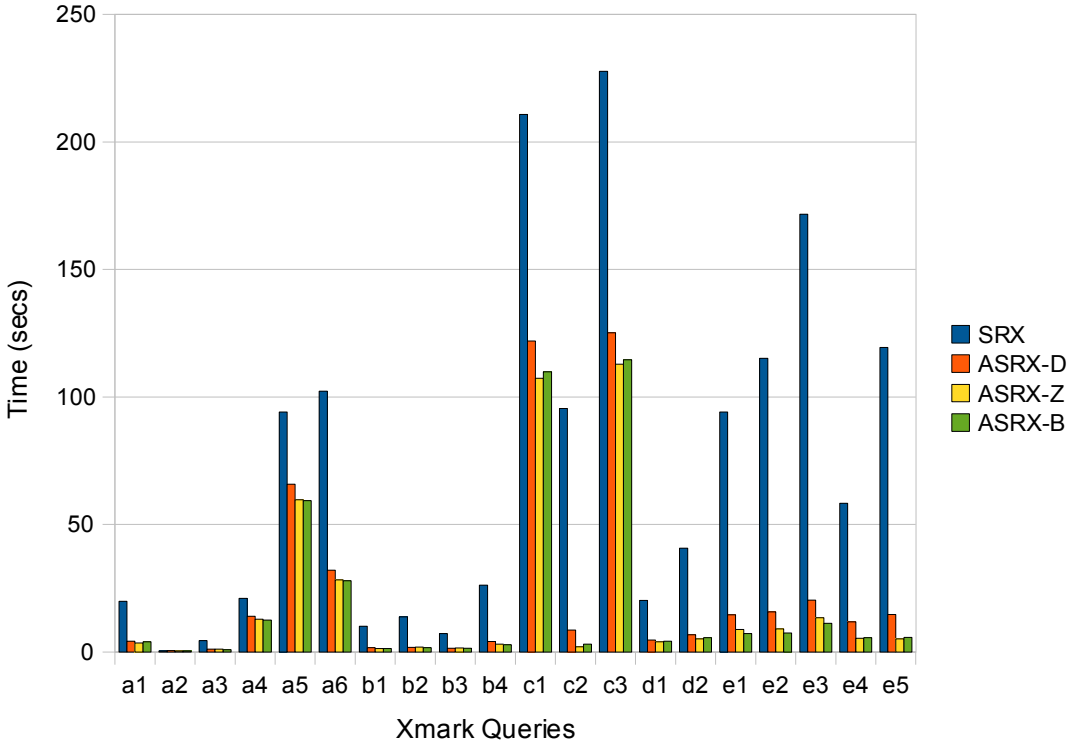
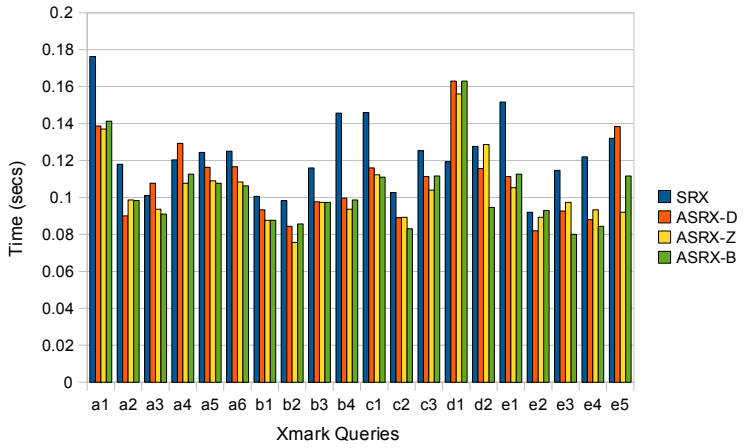
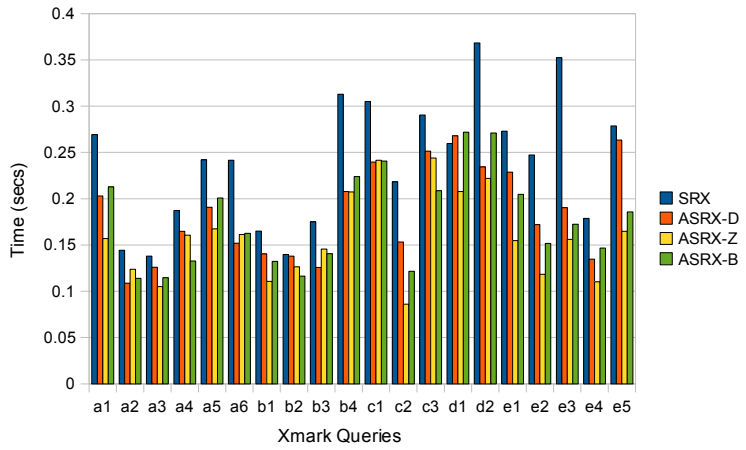


Figure 6.6: ASRX query evaluation for Xmark100 dataset

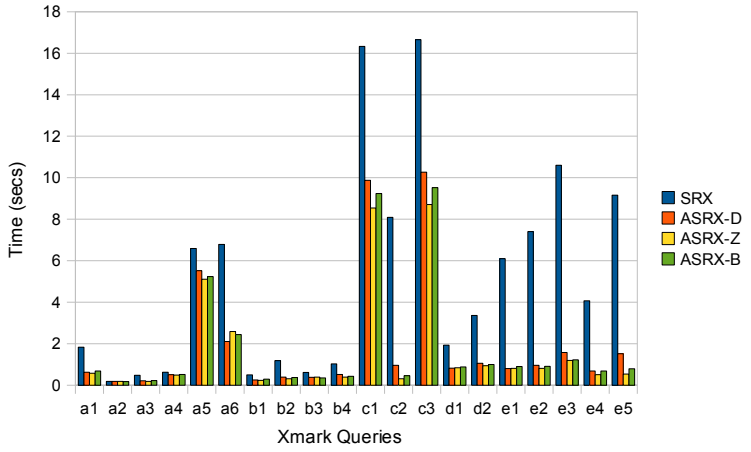
Finally, we present our experimental results for the smaller Xmark datasets. The reported evaluation times of ASRX (all variants) and SRX are shown in Figure 6.7. We observe that for the smallest Xmark dataset, despite the significant Path Stripe node and size compression accomplished, query evaluation time is not proportionally improved; in contrast, for some queries, we observe that compression rather hurts query performance. This occurs due to the low Path Stripe storage requirement for the explicit storage scheme; the time required for Stripe node decompression for ASRX is now comparable to the time saved from accessing the compressed Stripes instead the required Stripes at the explicit storage scheme. As the Xmark size increases though, the decompression time becomes insignificant compared to the time savings due to the compressed Stripes and as a result, ASRX performs better than SRX. The larger the size



(a) Xmark0.1



(b) Xmark1



(c) Xmark10

Figure 6.7: ASRX query evaluation for Xmark0.1, Xmark1 and Xmark10 datasets

Mbench Scaling Factor	Path Stripe Cardinality (Nodes)				CR_n		
	SRX	ASRX-D	ASRX-Z	ASRX-B	ASRX-D	ASRX-Z	ASRX-B
0.1	67697	213	68	56	317.83	995.54	1208.88
1	738984	2059	360	189	358.9	2052.73	3909.97
10	7291959	20284	3348	1605	359.49	2178	4543.28

Table 6.3: Compression effect on Path Stripe cardinality for Mbench datasets

Mbench Scaling Factor	Path Stripe Size (Pages)				CR_p		
	SRX	ASRX-D	ASRX-Z	ASRX-B	ASRX-D	ASRX-Z	ASRX-B
0.1	440	91	37	32	4.84	11.89	13.75
1	4456	711	136	74	6.27	32.76	60.22
10	43635	6816	1135	548	6.4	38.44	79.63

Table 6.4: Compression effect on Path Stripe size for Mbench datasets

of the dataset, the bigger the compression benefit for query evaluation performance.

6.6.2 Mbench

We move on to the impact of the structural agnostic compression on the Mbench dataset. In Table 6.3, we provide the Path Stripe cardinalities and the achieved node compression ratios for all proposed compression methods. All three compression methods achieve high node compression ratios. As in the case of the Xmark dataset, ASRX-D achieves the lowest node compression ratios compared to the two other compression methods, ASRX-Z always performs better than ASRX-D while ASRX-B achieves the best compression ratios. What differs though, is that while the node compression ratio achieved by ASRX-D is comparable to the one achieved for Xmark, ASRX-Z and ASRX-B perform 2-5 and 2-7 times better, respectively, for the Mbench dataset. In other words, ASRX-Z and ASRX-B accomplish significantly larger node reduction for the Mbench dataset compared to the reduction achieved for the Xmark dataset. This is due to the fact that Mbench datasets mostly consist of "eNest" elements and that all leaf eNest elements share the same structure. Thus, for the Stripes containing leaf elements, $end - start$ offset values will be highly repeated and both compression techniques can efficiently deal with repeated occurrences. For the Mbench dataset, the average cardinality of the compressed Stripes for ASRX-D, ASRX-Z and ASRX-B is

less than 0.3%, 0.1% and 0.05%, respectively, of the Path Stripe cardinality of the explicit storage scheme.

In addition to Path Stripe cardinality, in Table 6.4, we report Path Stripe storage information for the agnostic compression storage scheme. As in the case of the Xmark dataset, the trends for size compression ratios follow the node compression ratios: ASRX-D always produces the worst size compression ratios (6.4:1 for for the largest of Mbench dataset, Mbench10.), while ASRX-Z achieves up to a 6 times better compression ratio compared to ASRX-D. Finally, ASRX-B always achieves the best size compression ratio, up to 2 times better than the one achieved by ASRX-Z. In Figure 6.8, the impact of all three compression methods on Path Stripe storage is shown. We verify that:

- Agnostic compression has a bigger impact on Path Stripe storage of the Mbench dataset compared to the Xmark dataset. Regardless of the compression method used, the size of Path Stripes is reduced by *at least* 80%.
- As in the case of the Xmark dataset, the compression achieved for the smallest Mbench dataset is much worse than the one achieved for the larger datasets.
- Agnostic compression using the Dewey encoding (ASRX-D), reduces the original Path Stripe size (explicit scheme) up to 85%.
- Agnostic compression using the Zlib compression algorithm (ASRX-Z), always produces significantly better compression ratios for the Mbench dataset. Compared to the Dewey encoding, the total Path Stripe size is further reduced by 10%-15%.
- Agnostic compression using the Bzip compression algorithm (ASRX-B), always produces the best compression ratios for the Mbench dataset. Compared to ASRX-Z, it further reduces total Path Stripe size by a factor of 2 (for scaling factors 1,10).

We now present the node and size reduction of the Stripes considered for each of the Mbench queries in isolation. The results for the largest of the Mbench datasets (sf=10, size=5GB) are depicted in Figure 6.9. As seen in Figure 6.9(a), for the vast majority of the Mbench queries, the required Stripes that need to be accessed, contain 20% to 50% less nodes from those that are stored using the explicit storage scheme. Regardless

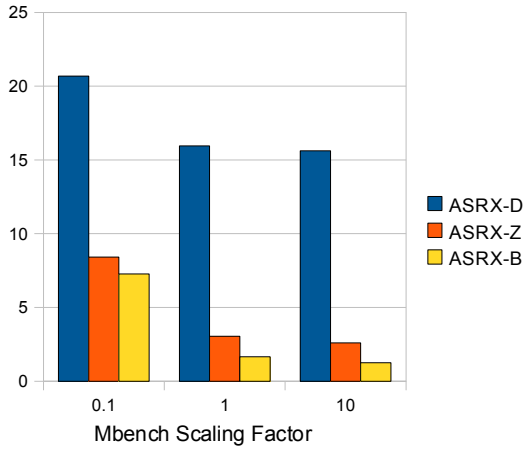
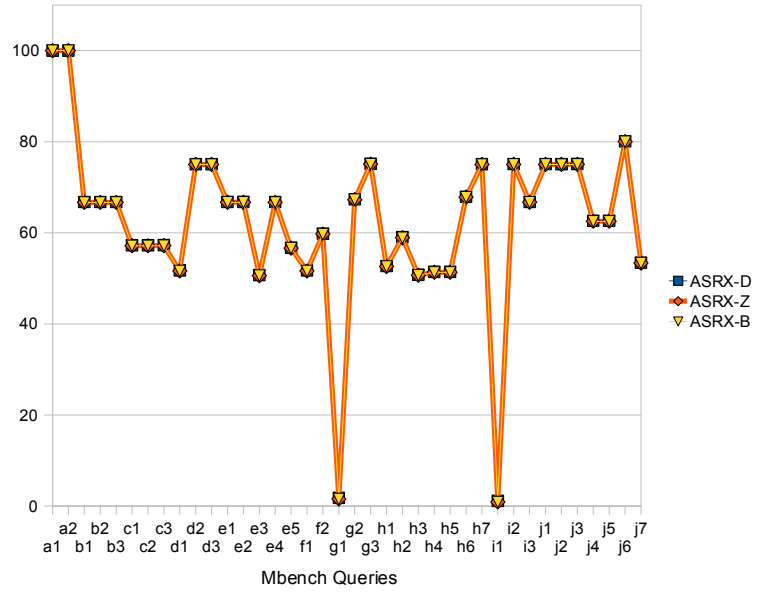


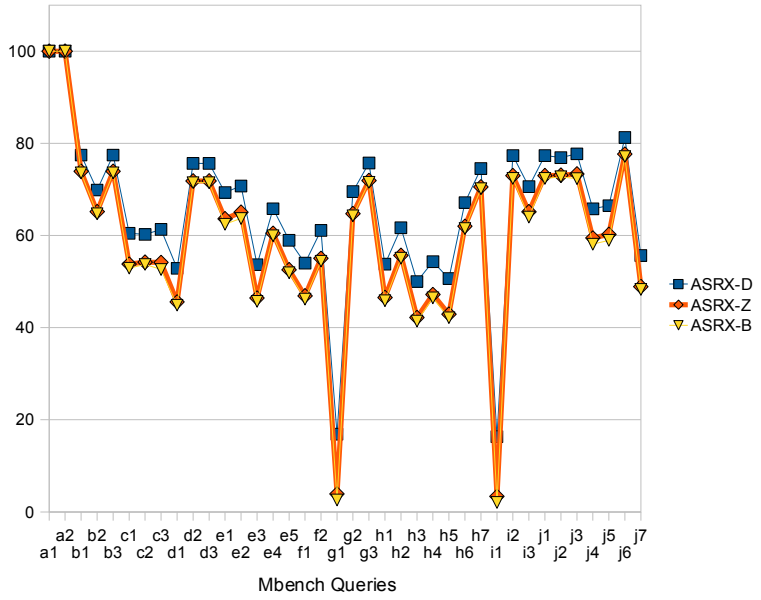
Figure 6.8: Path Stripe size compression for Mbench datasets

of the compression method used the average node reduction for all Mbench queries is around 35%. This contradicts though with the node compression ratios achieved for the Mbench dataset; the node compression ratios achieved for the Xmark dataset were much worse compared to those achieved for the Mbench dataset (especially for ASRX-Z and ASRX-B), however the node reduction percentage per query was higher to that achieved for Mbench. This occurs due to the fact that for most of Mbench queries, a significant amount of Attributes Stripes are involved, for which no compression occurs. Thus, it is normal for the overall (Path, Attribute and Value) node reduction percentage to be reduced if a significant amount of nodes is not compressed at all. As far as the Path Stripe storage is concerned, in Figure 6.9(b), the size reduction per Mbench query is shown which is proportional to the node reduction. The average size reduction is 34.6% for ASRX-D and increases to 41% for ASRX-Z and ASRX-B.

We now compare the query evaluation performance of the agnostic compression storage scheme to the one of the explicit storage scheme. In Figure 6.10, we present the evaluation times for the largest of the Mbench datasets, Mbench10. As in the case of the Xmark dataset, our query engine over the agnostic compression storage scheme outperforms the one over the explicit storage scheme. For ASRX-D, query evaluation performance is improved by 33% on average compared to SRX. ASRX-Z and ASRX-B perform even better due to the higher compression ratio; ASRX-Z achieves an average speedup of 40% while ASRX-B's performance is improved by 38.3%, compared to SRX. For the majority of Mbench queries, ASRX-Z performs better than ASRX-B (and ASRX-D) and, in general, ASRX-Z has the lead in query evaluation performance for the



(a) Node reduction



(b) Size reduction

Figure 6.9: Agnostic compression effect for Mbench queries

Mbench dataset, despite the fact that ASRX-B achieves better compression ratios.

We conclude with our experimental results for the smaller Mbench datasets, as shown in Figure 6.11. For the smallest Mbench dataset and despite the small dataset size, the compression benefits are evident; to a smaller extent though and not for all compression methods. ASRX-D, for instance, for more than half of the Mbench queries,

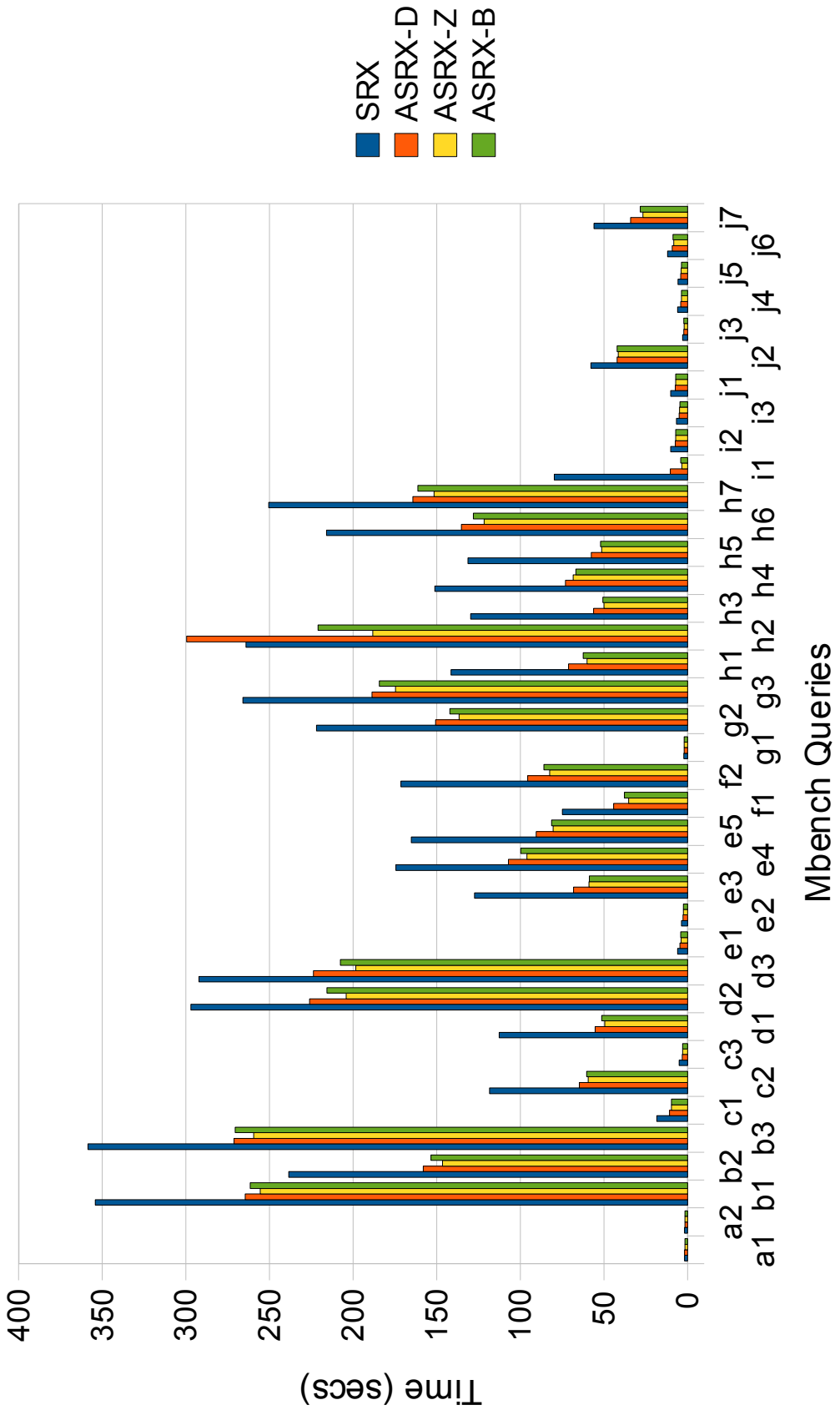
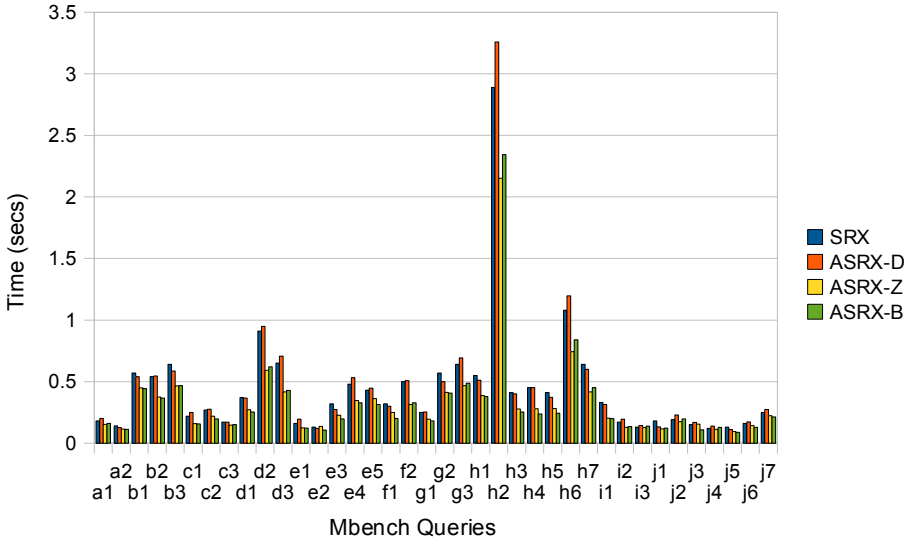
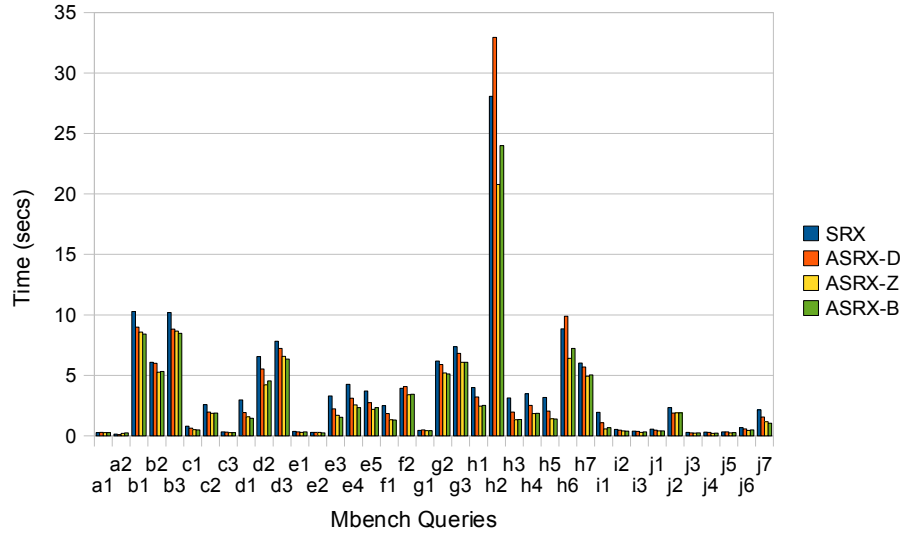


Figure 6.10: ASRX query evaluation for Mbench10 dataset

performs worse than SRX. On the other hand, ASRX-Z and ASRX-B achieve a time speedup of 23% and 25% respectively. For the mid-size Mbench dataset, Mbench1, the compression benefits are clearer; all three compression methods perform better than SRX, achieving time speedups of 14% (ASRX-D), 27% (ASRX-Z) and 25% (ASRX-B).



(a) Mbench0.1



(b) Mbench1

Figure 6.11: ASRX query evaluation for Mbench0.1 and Mbench1 datasets

DBLP Scaling Factor	Path Stripe Cardinality (Nodes)				CR_n		
	SRX	ASRX-D	ASRX-Z	ASRX-B	ASRX-D	ASRX-Z	ASRX-B
1	2609007	6879	3447	2685	379.27	756.89	971.7
5	13045027	33979	16773	12915	383.91	777.74	1010.07
10	26090052	67875	33408	25676	384.38	780.95	1016.13

Table 6.5: Compression effect on Path Stripe cardinality for DBLP datasets

DBLP Scaling Factor	Path Stripe Size (Pages)				CR_p		
	SRX	ASRX-D	ASRX-Z	ASRX-B	ASRX-D	ASRX-Z	ASRX-B
1	15758	2378	1149	896	6.63	13.71	17.59
5	78153	11476	5336	4041	6.81	14.65	19.34
10	156144	22828	10566	7984	6.84	14.78	19.56

Table 6.6: Compression effect on Path Stripe size for DBLP datasets

6.6.3 DBLP

We now proceed to the impact of the structural agnostic compression on the DBLP dataset. As seen in Table 6.5, the node compression ratios achieved by the proposed compression methods, present similar characteristics as for the Xmark and Mbench datasets: ASRX-D always achieves the lowest node compression ratios compared to the two other compression methods, ASRX-Z always performs better than ASRX-D and ASRX-B achieves the best compression ratios. Node compression ratios are improved by a factor of 2 for ASRX-Z and by a factor of 3 for ASRX-B, compared to ASRX-D. The average cardinality of the compressed Stripes for ASRX-D, ASRX-Z and ASRX-B is less than 0.26%, 0.13% and 0.1%, respectively, of the Path Stripe cardinality of the explicit storage scheme.

The compression impact on Path Stripe storage is proportional to the compression impact on Path Stripe cardinality. As seen in Table 6.6, the size compression ratio achieved for ASRX-Z is improved by a factor of 2 compared to ASRX-D, which always produces the worst size compression ratios. Likewise, the size compression ratio achieved for ASRX-B is improved by a factor of 3 compared to ASRX-D. The impact of all three compression methods on Path Stripe storage is also depicted in Figure 6.12, where the normalised Path Stripe size for each of the proposed compression methods is shown. This time, the size compression ratio is relatively constant to the size of the XML dataset. ASRX-D reduces the Path Stripe size by 85%, while ASRX-Z and ASRX-B

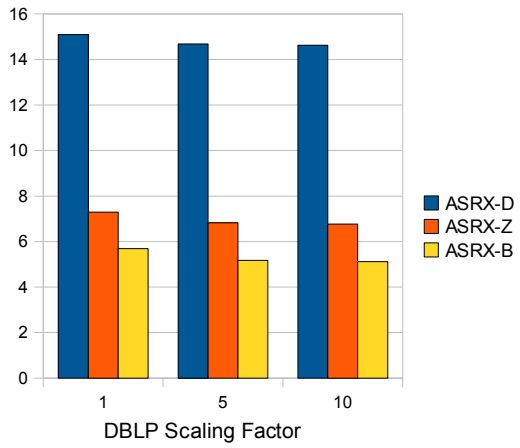
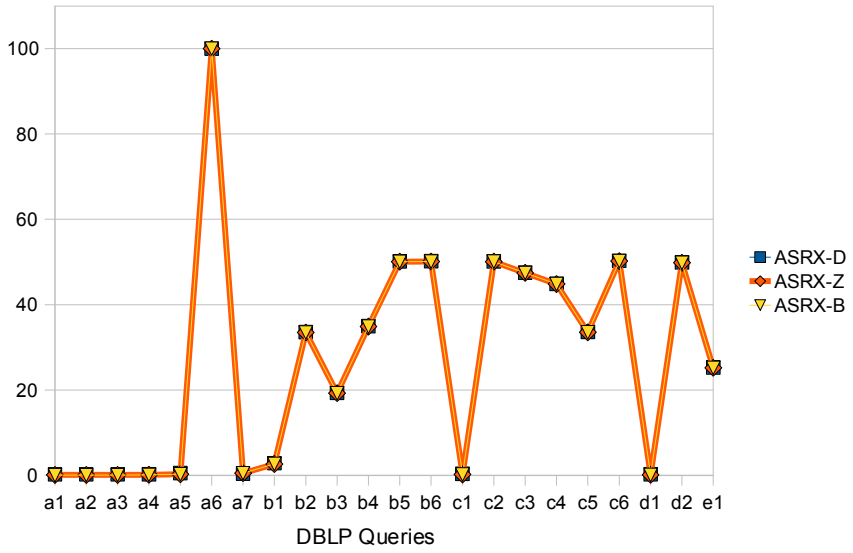


Figure 6.12: Path Stripe size compression for DBLP datasets

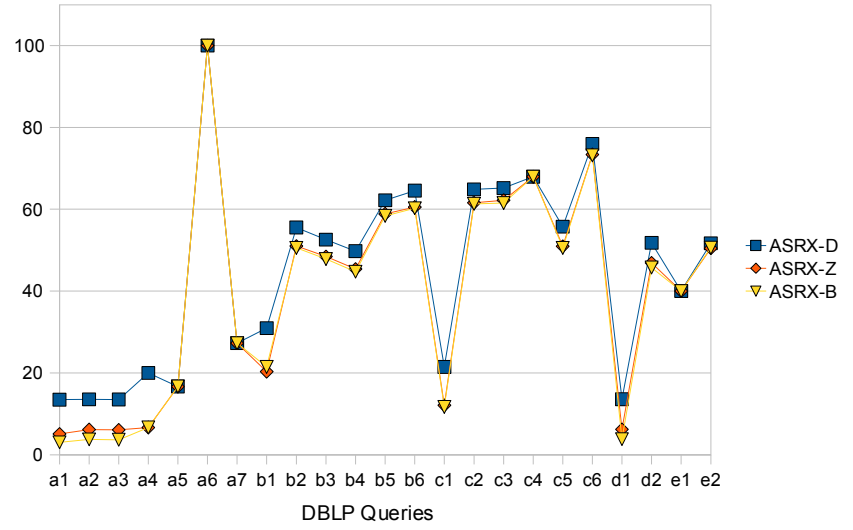
achieve a size reduction of 92% and 95% respectively.

We now turn our attention to the compression effect in terms of node and size reduction of the Stripes that are considered for each of the DBLP queries. The results for the largest of the DBLP datasets (sf=10, size=1GB) are depicted in Figure 6.13. As seen in Figure 6.13(a), for almost half of the DBLP queries, the required Stripes that need to be accessed, contain less than 1% of the nodes that are stored in the corresponding Stripes using the explicit storage scheme. Regardless of the compression method used the average node reduction for the DBLP queries is around 72.5%. While the node compression ratio is similar for all three compression methods, this does not hold for the size compression ratio; The average size reduction for ASRX-D is 55.3%, while ASRX-Z and ASRX-B further reduce Stripe size by 5% compared to ASRX-D.

We now compare the query evaluation performance of the agnostic compression storage scheme to the one of the explicit storage scheme. In Figure 6.14(c), we present the evaluation times for the largest of the DBLP datasets, DBLP10. As in the case of the Xmark and Mbench datasets, the query engine over the agnostic compression storage scheme outperforms the one over the explicit storage scheme. What is interesting in this case, is that all three compression methods have similar impact on query evaluation. For ASRX-D, query evaluation performance is improved by 45.7% on average compared to SRX. ASRX-Z and ASRX-B perform slightly better (on average); ASRX-Z achieves an average speedup of 47.5% while ASRX-B's performance is further improved by 1% on average, compared to ASRX-Z. The distribution of DBLP



(a) Node reduction



(b) Size reduction

Figure 6.13: Agnostic compression effect for DBLP queries

queries that each of the ASRX variants performs better than the other two variants is as follows: ASRX-D, ASRX-Z and ASRX-B perform better for almost 22%, 48% and 30% of the DBLP queries respectively. Note, that this is the first time that the Dewey-based compression has such an impact on query performance. As far as the other two variants are concerned, it is hard to say which is dominant. ASRX-B achieves better compression ratios and larger average query evaluation speedups, however, these are too close to the ones achieved by ASRX-Z. ASRX-Z, on the other hand, performs better

than ASRX-B for more than half of the DBLP queries.

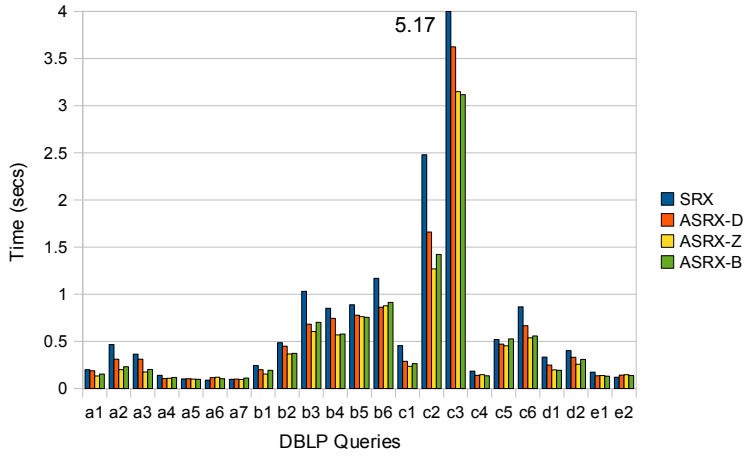
We finally conclude with our experimental results for the smaller DBLP datasets, as shown in Figure 6.14(a) and 6.14(b). As usual, the compression impact is not as evident for the smallest dataset due to the small dataset size. Nevertheless, a significant time improvement is observed: ASRX-D, ASRX-Z and ASRX-B achieve a time speedup of 15%, 26% and 23% respectively. For the mid-size DBLP dataset, DBLP5, the compression benefits become clearer; all three variants of ASRX perform better than SRX by a factor of 2 on average.

6.6.4 Preferred Compression Method

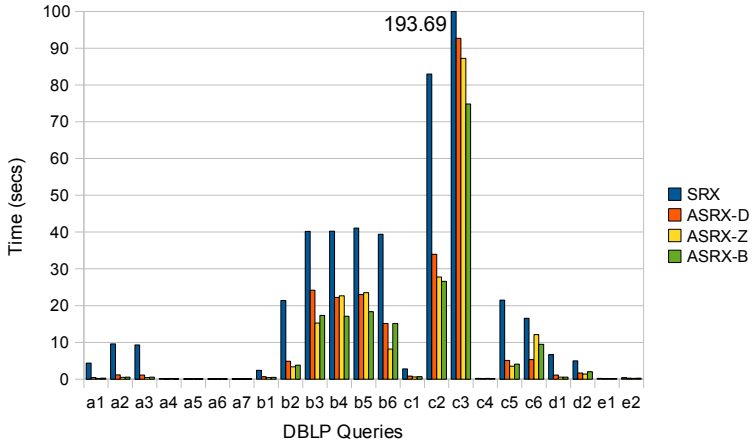
We now discuss the results of the three alternative options for compressing Path Stripe node chunks.

The Dewey encoding is by far the worst option for both compression and querying. As already presented, for each of the tested datasets, ASRX-D produces the worst node and size compression ratios. Especially for the Mbench and DBLP datasets, its compression effect is poor compared to the other two alternatives. This is also reflected on query evaluation times; for the vast majority of all tested queries, ASRX-D performs worse than any of the other two alternatives.

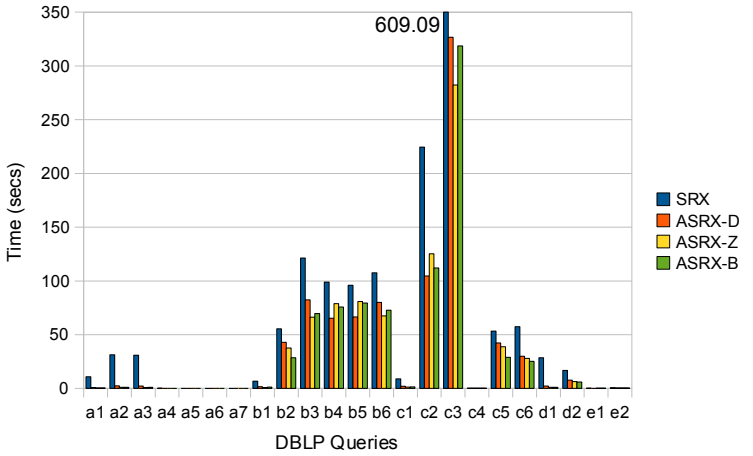
Among ASRX-Z and ASRX-B, there is no clear winner. If our metric is the compression effect, then ASRX-B outperforms ASRX-Z, as it reduces both overall Stripe cardinality and size. However, during query evaluation, ASRX-B's advantage over ASRX-Z is not evident. This happens because if we consider each query in isolation, the size reduction of the selected for query evaluation Stripes, achieved by any of ASRX-B or ASRX-Z is almost at the same level. This is also a side-effect of the fact that compression only concerns Path Stripes, while a significant number of Attribute/Value Stripes can also be involved in a query. In most cases, ASRX-B further reduces a relatively small number of Stripe pages compared to ASRX-Z. However, as already described in Section 6.2.3, bzip2 is considerably slower than zlib during decompression. Thus, in many cases, the small I/O overhead that ASRX-Z pays compared to ASRX-B, is counterbalanced by the computational speedup during decompression. Overall, if we are mostly interested in query evaluation performance, then the zlib compression method is preferred over bzip2. According to our experimental results, ASRX-Z performs better than ASRX-B in a larger number of queries, while it performs consistently better for the Mbench dataset.



(a) DBLP1



(b) DBLP5



(c) DBLP10

Figure 6.14: ASRX query evaluation for DBLP1, DBLP5 and DBLP10 datasets

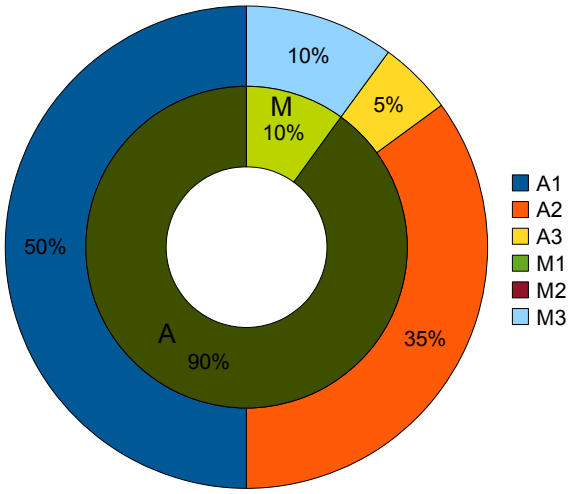


Figure 6.15: Xmark query distribution for ASRX compared to MDB (inner) and both MDB and SRX (outer) results

6.6.5 Comparison with MDB

We now compare our evaluation engine prototype over the agnostic compression storage scheme, ASRX, to the state-of-the-art in XML query processing, MDB. For the comparison with MDB, we selected the zlib compression method, since we are mostly interested in query performance and not in the achieved compression; for this section, ASRX is a synonym to ASRX-Z. The comparison is structured in a similar manner to the comparison of CSRX with MDB. To that end, we also consider the evaluation results of SRX, which provides the baseline case for Path Stripe storage. In addition, for each of the tested datasets, we divide the queries in two main categories:

- Query Category *A*, which consists of all queries for which ASRX outperforms MDB, and
- Query Category *M*, which consists of all queries for which ASRX performs worse than MDB.

Both query categories are further divided into sub-categories as described in Section 5.6.5.

6.6.5.1 Xmark

The evaluation results of ASRX, SRX and MDB for the largest Xmark dataset (sf=100, size=11GB), are displayed in Figure 6.16. We also present the distribution of the

Xmark queries in the main query categories and sub-categories in Figure 6.15. First of all, we remind the reader that for all Xmark queries (except a2), ASRX outperforms SRX by a large factor (73.5%). Even for query a2, the resulting difference is insignificant and has nothing to do with the compression impact since the evaluation process (for the PRO evaluation setup), merely involves Data Stripes and thus no compression is applied. Thus, we can safely assume that for all Xmark queries, ASRX is the clear winner compared to SRX.

Regarding the comparison of ASRX to MDB, ASRX outperforms MDB for 90% of the Xmark testbed (query category *A* - 18/20 queries). For this category, ASRX achieves a performance speedup of 73.5% on average[1]. Query sub-category A_1 contains queries a1, a6, b4-c2 and e1-e5 and consists of 50% of all Xmark queries. For such queries, agnostic compression enables ASRX to outperform MDB, in addition to SRX. Query sub-category A_2 contains queries a3-a4, b1-b3 and d1-d2, that is 35% of all Xmark testbed queries. For the queries that SRX performs better than MDB, the added agnostic compression effect, further improves query evaluation performance, producing even better evaluation results compared to MDB. Query sub-category A_3, consists of query a2, on which, as described, ASRX and SRX perform (almost) the same; both systems outperform MDB.

On the other hand, MDB performs better than ASRX only for two Xmark queries. These are queries a5 and c3 that both contain very selective value-based predicate expressions. Even for these two queries though, the compression effect is evident from the comparison of ASRX to the SRX results.

6.6.5.2 Mbench

We now proceed to the evaluation results for the largest Mbench dataset (sf=10, size=5GB), as shown in Figure 6.17. We have already described that ASRX (*i.e.,* ASRX-Z) compression has a huge impact on query evaluation; ASRX outperforms SRX for *all* Mbench queries by 40% on average.

Apart from SRX though, ASRX also outperforms MDB for the majority of Mbench queries. In detail, for 92% of the Mbench query testbed (35/38 queries), ASRX provides an average evaluation speedup of 79%[2]. As in the case of CSRX, however, for a large

[1]Excluding the result for query d2, that MDB fails to evaluate.

[2]Excluding the results for queries h1-h7, that MDB fails to evaluate.

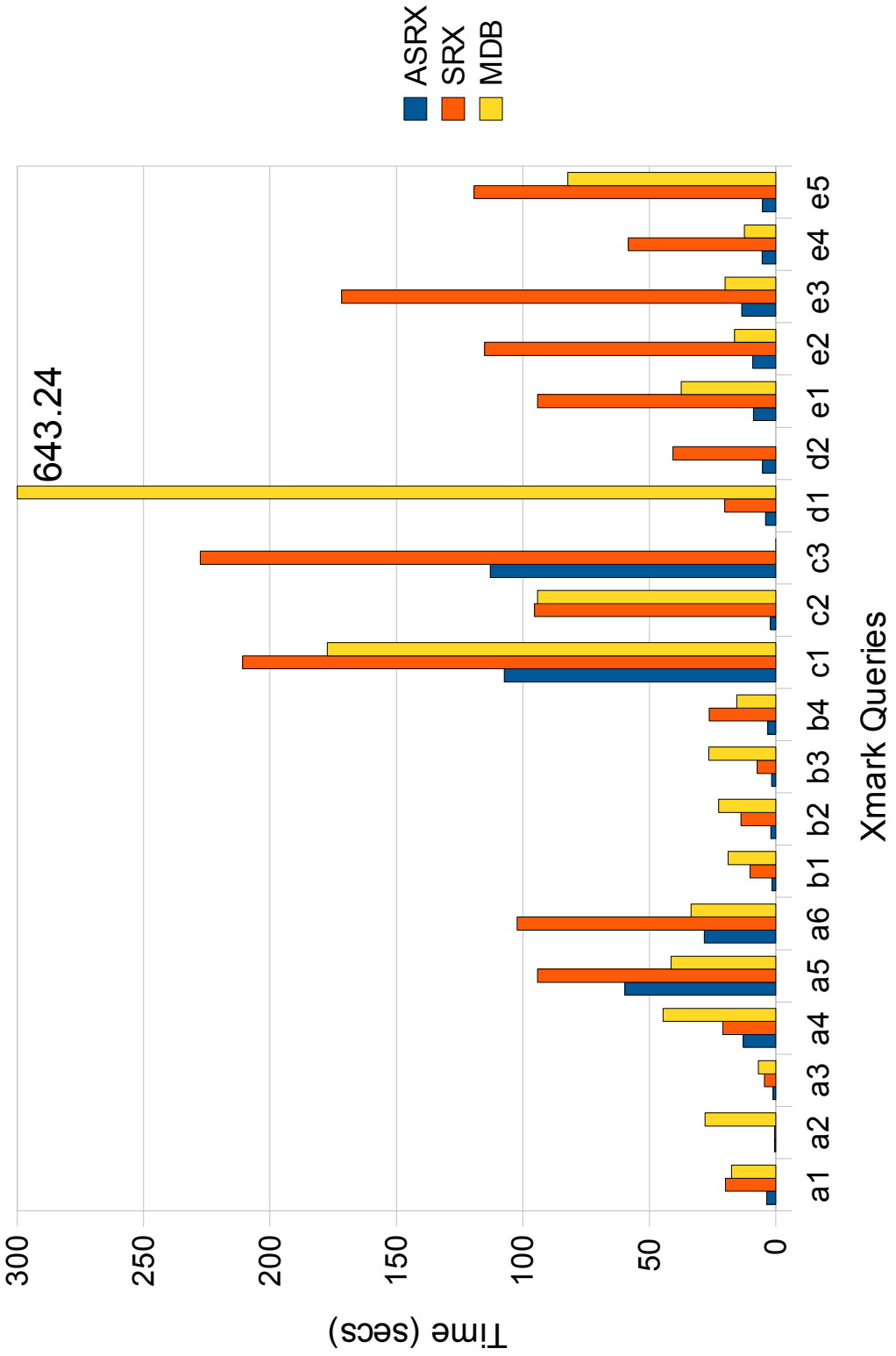


Figure 6.16: ASRX and MDB Comparison for the Xmark100 dataset

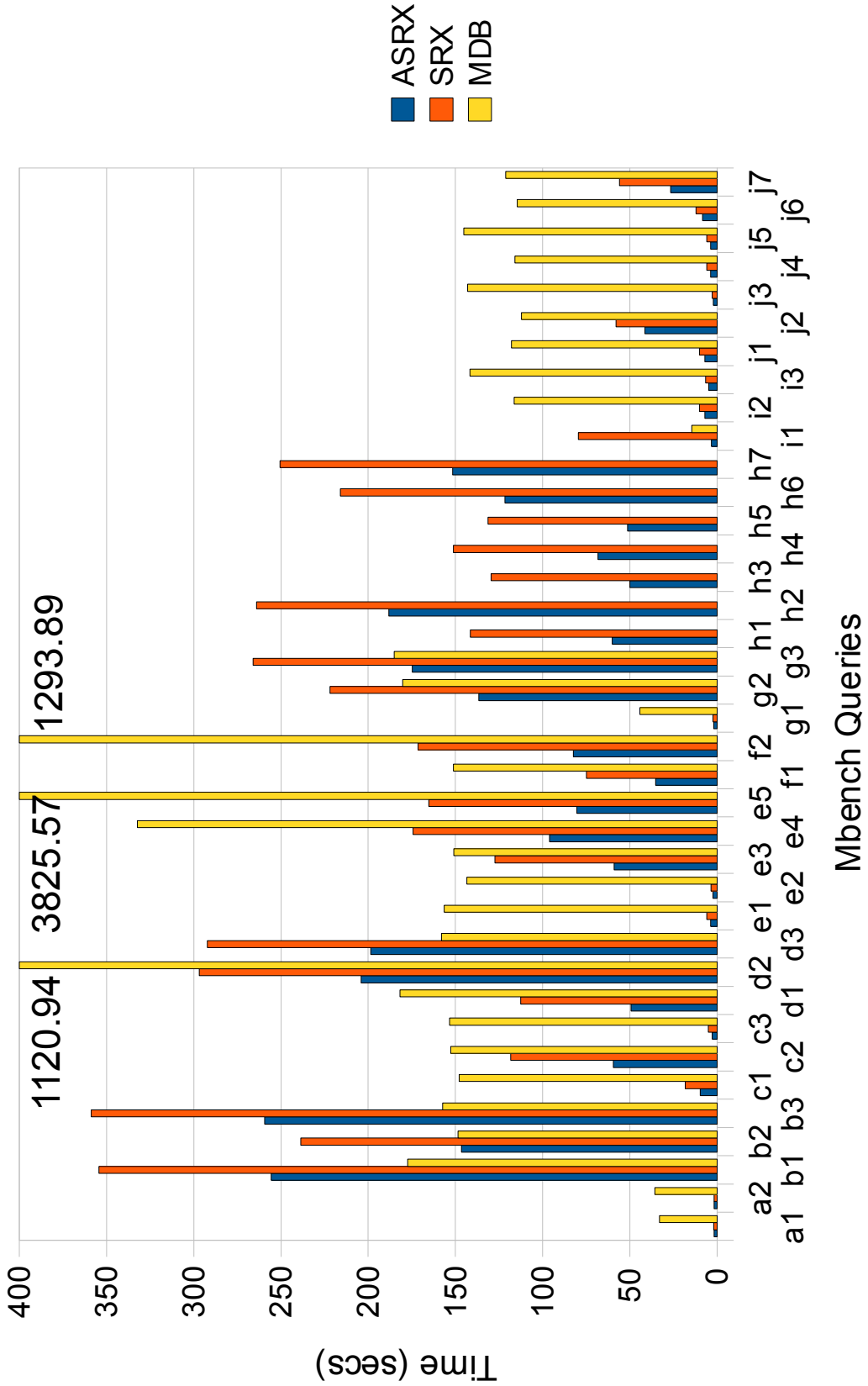
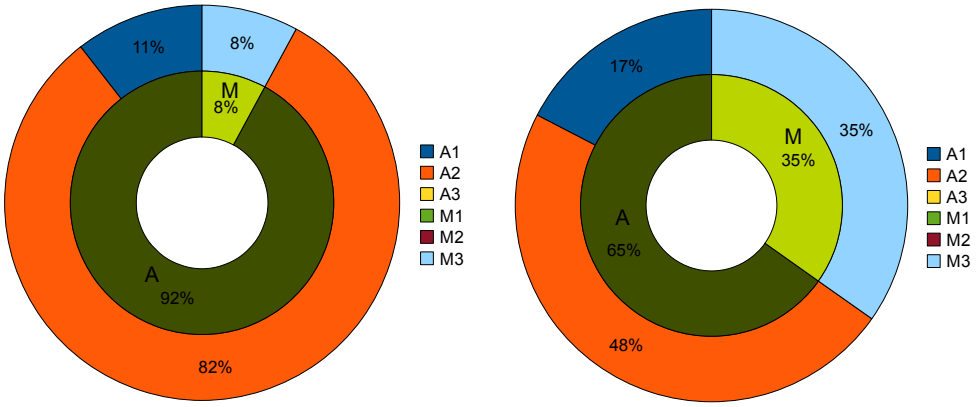


Figure 6.17: ASRX and MDB Comparison for the Mbench10 dataset



(a) Mbench (b) DBLP

Figure 6.18: Query distribution for ASRX compared to MDB (inner) and both MDB and SRX (outer) results

part of query category A, SRX itself outperforms MDB, and thus the compression effect of ASRX further improves evaluation times over SRX. This occurs for queries a1-a2, c1-d2, e1-g1, h1-h7 and i2-j7, (sub-category A_2), that amounts to 82% of the Mbench query testbed. For the rest of category A, queries b2, g2-g3 and i1, it is the added compression effect that boosts ASRX query evaluation, so that it performs better than MDB.

MDB merely outperforms ASRX for 8% of the query testbed (3/38 Mbench queries). For these queries, namely queries b1, b3 and d3, MDB performs 30% faster than ASRX on average, due to very selective value-based predicates. Nevertheless, the performance of SRX for those queries is significantly improved due to the ASRX compression impact.

6.6.5.3 DBLP

We finally present the evaluation results for the largest DBLP dataset (sf=10, size=1GB), as shown in Figure 6.19. Recall that ASRX (with zlib compression) performs better than SRX for all DBLP queries by 47.5% on average.

Regarding the performance comparison between ASRX and MDB, we notice that the query distribution in categories based on the comparison results is exactly the same as the one from the comparison of CSRX to MDB. Thus, for the DBLP dataset, both the agnostic and tree-sharing compression schemes have the same impact on query evaluation performance. In fact, the evaluation results from both systems produce the

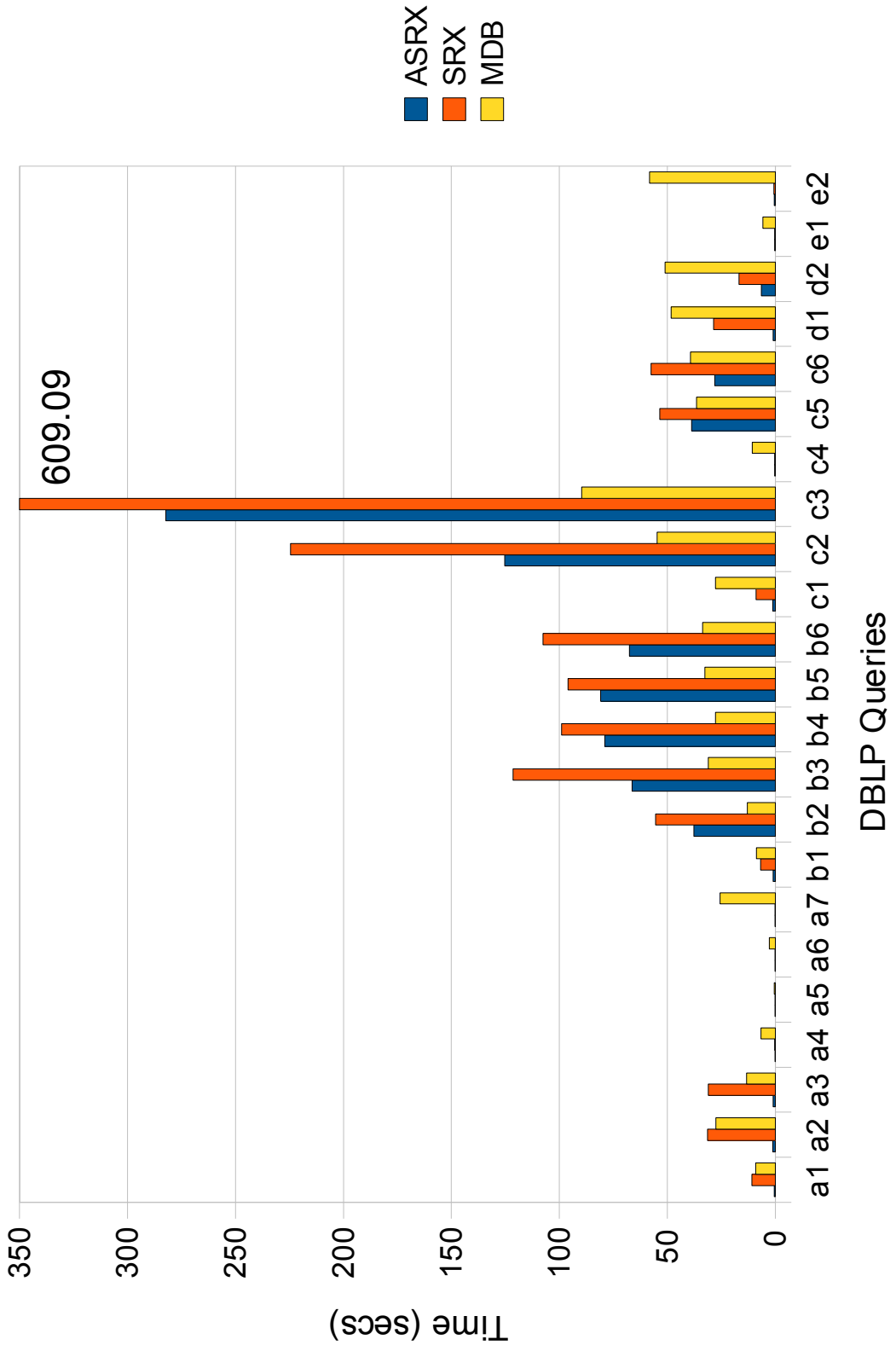


Figure 6.19: ASRX and MDB Comparison for the DBLP10 dataset

same query distribution in sub-categories.

ASRX outperforms MDB for 65% of the DBLP query testbed (15/23 DBLP queries) by 90% on average. Category A_1 (17% of the DBLP testbed), includes queries a1-a3 and c6, where the added compression effect results in better query performance than that of MDB. Similarly, category A_2 (48% of the DBLP testbed), includes queries a4-b1, c1, c4 and d1-e2, where the added compression effect further improves SRX query evaluation performance that already outperforms MDB.

For the rest of the DBLP query testbed (35%), namely queries b2-b6, c2-c3 and c5, ASRX underperforms by 53% on average compared to MDB. Although ASRX compression increases the query performance with respect to SRX, this is still worse than the query performance of MDB. Recall that all these queries contain very selective value-based predicates that dominate the overall query evaluation performance.

6.7 Related Work

We presented two storage schemes that compress the structural part of a striped XML document. The tree-sharing compression storage scheme, described in Chapter 5, works by identifying structural regularities of an XML document. Similar, repetitive subtree structures are condensed into a single instance and this is reflected by compressed Path Stripe nodes in the corresponding set of Path Stripes. In addition, we proposed the agnostic compression storage scheme, described in Chapter 6, which employs compression on the structural part of a striped XML document regardless of its characteristics. Both approaches apply compression on the document structure only, *i.e.,* on the structure of a striped XML document, as this is described by the explicit storage scheme in Chapter 4. While we focus on the compression of the document's structure, there exists a significant amount of work dealing with XML data compression.

Compressing XML documents received much attention even since the early years that XML emerged. The flexible, self-describing representation of XML data results in size inflation since tags are used to describe data and these are largely repeated. To deal with the data inflation problem, a large number of XML compressors/decompressors has been proposed. These can be classified based on their awareness of the specific XML format. To that end, there exist XML-oblivious and XML-conscious compressors. Since an XML document is a sequence of textual data, the idea of using general-purpose compressors was proposed. Indeed, the use of generic text compressors such

as bzip2 [89] and gzip [42] (Sections 6.2.2, 6.2.3), has resolved the size inflation problem to a big extent. However, XML-conscious compressors have also emerged as better results could be obtained if the self-describing XML format was taken into account. Apart from the data inflation problem, another important factor that was considered was whether a compressed format of an XML document would also allow querying. Based on that, XML-conscious compressors were divided in non-queryable or archival XML compressors and queryable XML compressors. We now present some of the approaches that are mostly related to our work. Many other XML compressors have also been proposed. For further details, please refer to [78, 86].

6.7.1 Archival XML Compressors

The focus of an archival XML compressor is to reduce the size of an XML document to the maximum extent possible. *XMill* [64] is the first implementation of an XML-conscious, archival compressor. The most important ideas of XMill, that have largely influenced many other subsequent compressors as well as XML stores (such as ours), are: (a) the separation of document structural information from data, and (b) the grouping of data items with related semantics into homogeneous containers. In XMill, the structural part of the document is extracted and element tags and attributes are encoded in a dictionary-based fashion. Each end-tag is replaced by a special code '/'. The document structure is stored in a separate structure container which is then compressed by a general text compressor, gzip, and is appended to the output file. Data values are grouped into containers based on their element/attribute label and type. While this is the default grouping, other alternatives can be specified such as the path-based grouping or any other grouping that is based on path expressions. The application of specialised semantic compressors per container is also possible, although it is user-guided. Each container is then compressed independently with gzip and appended to the output file. Grouping data values into semantically related containers effectively enhance the achieved compression by localising the data repetitions. Apart from gzip, other general text compressors can be used such as bzip2. XMill is a specialised, XML-conscious compressor; it consistently achieves better compression ratios than the XML-oblivious compressors. However, for evaluating queries over the compressed documents, these must be first fully decompressed.

6.7.2 Queryable XML Compressors

The focus of a queryable XML compressor is twofold: Since it is an XML compressor it strives for compression and thus to reduce the size of the original XML document. However, the compressed XML document must also be query-friendly, *i.e.,* it should allow queries to be processed on the compressed XML format. There is a trade-off between the achieved compression ratio and query evaluation performance. Queryable XML compressors can be further divided in two sub-categories:

6.7.2.1 Homomorphic Compression

This category includes compression approaches that result in a compressed XML instance that preserves the structure of the original document and is also an XML document.

XGrind [96] is the first homomorphic, XML-conscious compressor that allows querying without the need for full decompression of the compressed XML document. In XGrind, the compressed instance of an XML document is still an XML document, whose tags, attributes and textual values are replaced with proper encodings: Similar to XMill, the element tags and attribute names are encoded using dictionary encoding. However, unlike XMill, the textual values are compressed in isolation, using binary encodings for enumeration-type attribute values and non-adaptive, *context-free* Huffman encoding [54] for any other textual values. This, along with the fine granularity of textual value compression, enables XGrind to evaluate exact-match and prefix-match predicates directly on the compressed document instance. However, for handling partial-match or range predicates, XGrind needs to decompress the values that are related to the query predicates, since the Huffman encoding does not preserve order information. The degraded compression ratio of XGrind with respect to XMill is the side-effect of providing querying capabilities over compressed XML documents, avoiding full or even partial decompression. On the other hand, an important limitation of XGrind is that the pattern matching approach for query processing covers efficiently only a limited set of XPath queries, including child and attribute axes. For instance, the evaluation of a descendant axis location step expression will result in processing large and possibly irrelevant parts of the compressed instance.

Another homomorphic queryable XML compressor is *XPress* [75]. XPress shares many common characteristics with XGrind. Apart from preserving the original XML document structure, both approaches compress data values in isolation. In addition,

both compressors use Huffman and dictionary encoding for compressing textual and enumeration-type data, respectively. This enables the direct evaluation of exact-match and prefix-match predicates on compressed XML data. XPress departs from XGrind in that it employs an automated type inference mechanism for identifying the type of data values of each specific element. XPress can therefore apply proper encoding methods for each of its supported types. In addition to textual and enum values, XPress identifies numerical values which are compressed using the binary encoding in conjunction with the differential encoding. This encoding method preserves the order of the compressed data values and enables XPress to evaluate range predicates directly on the compressed data. Nevertheless, this is restricted on numerical values. For textual values, XPress still needs to perform partial decompression of the values that are relevant to the query predicate. However, the novel feature of XPress is its reverse arithmetic encoding scheme it employs for structural encoding. According to this, each label and label-path (path for short) is encoded as an interval of real numbers between $[0.0, 1.0)$, so that if a path p_2 is as suffix of path p_1, then p_2's interval contains p_1's interval. XPress achieves better compression compared to XGrind. It also provides faster query evaluation, as it is able to directly identify node containment relationships; instead of matching encoding paths for each encountered element, as in the case of XGrind, it simply checks containment among path intervals.

6.7.2.2 Non-homomorphic Compression

We now present the non-homomorphic XML compressors. This category includes compression approaches that are similar to XMill (although it is a non-queryable compressor), in that they separate structure from data. This category is of great interest since our proposed storage model also relies on this decomposition approach.

XQzip [27] compresses the structure of an XML document by constructing a main memory structure, the *Structural Indexing Tree (SIT)*, which is based on the partitioning of equivalent paths; The SIT merges subtrees containing the exact same set of paths and thus effectively removes all duplicate structures. The SIT belongs to the family of structural summaries (see Section 1.1.3.1), but especially resembles the compressed skeleton in [20, 19]. One significant difference is that in the compressed skeleton, the global ordering of an XML node is preserved by the extensive use of skeleton edges. In the SIT, global ordering is not explicitly preserved but can be re-constructed from the unique ids assigned to the original XML nodes. As a result, the SIT is expected to have a smaller size than the compressed skeleton. On the other hand, for providing

document order, extra sorting is needed. For instance, the original document can be reconstructed from the compressed skeleton by a depth-first traversal. If the SIT is used instead, the decompressed nodes need further sorting on their ids to restore document order. For the textual data values of an XML document, these are compressed into distinct data containers, based on their label value and thus semantically related data are grouped together. To that end, XQzip departs from the XMill approach that compresses a data container as a whole, but also from the homomorphic approaches that compress each data value in isolation. In XQzip, each data container is further divided into smaller blocks that are individually compressed with a general text compressor, gzip. The selection of the block size is important; it provides a trade-off between the achieved compression ratio and the decompression overhead incurred during query evaluation. The block-based compression degrades the overall compression ratio of a data container, when compared to the Xmill approach, however it enables efficient querying since the full decompression of the container is avoided. In XQzip, with the use of the SIT index, the query processing engine can select a subset of data blocks to decompress and process. Despite that, the decompression time still dominates the overall querying time. To avoid the extra decompression cost of data blocks that are extensively reused, XQzip employs an LRU buffer pool for caching recently decompressed data. The queries addressable by XQzip belong to XPath 1.0 [32], enhanced with the grouping operator in the return step. XQzip achieves comparable compression to non-queryable XML compressors like XMill, while at the same time achieves better performance than queryable approaches *e.g.,* XGrind. However, the performance results of XQzip come at the cost of main memory-only usage and availability. The SIT index is a main memory summary and while in many cases it is much smaller than the original document tree, there is no guarantee of its maximum size. In addition, each compressed node stores the ids of the original nodes of its extent and thus, we expect the SIT index to require a significant amount of memory for encoding large XML documents. Apart from the SIT index, a buffer pool is used for caching recently decompressed data. This way, XQzip essentially causes double buffering of data containers. When a compressed data-block is requested from the disk, it is fetched in main memory and then buffered either explicitly by a dedicated buffer pool or implicitly by the operating system. Once the data-block is decompressed, the (decompressed and thus larger) block is also buffered by XQzip to avoid the extra decompression cost. Thus, the overall evaluation requires high availability of system resources and in particular system memory.

XCQ [79] is a schema-aware XML-conscious compressor. Similarly to XMill, XCQ separates the structural information from the textual information. For the former, instead of storing a complete structure stream or tree, it constructs a structure stream that only encodes information that cannot be derived from the DTD. This may include the occurrence (or not) of an optional element or the number of repetitions for a Kleene-star operator within the DTD. For the latter, XCQ uses a partitioned path-based grouping (PPG) to decompose textual information to a set of data streams; a data stream is created for each DTD path. Like in XQzip, data streams are then partitioned into blocks that can be independently compressed using a general text compressor. In addition, XCQ uses a minimal indexing scheme on the compressed blocks, the *Block Statistics Signature* so that during query evaluation, only the blocks that contain information relevant to the query, will be decompressed. Therefore, XCQ employs partial decompression. XCQ supports the evaluation of a subset of XPath 1.0 [32] (enriched with aggregation) queries over compressed XML data. One of the advantages of XCQ is that since all redundant structural information is already stored in a DTD, the achieved compression is comparable to, and in many occasions better than, the compression achieved by the non-queryable XML compressor, XMill. At the same time, XCQ provides better compression than homomorphic queryable approaches, such as XGrind. On the other hand, the obvious limitation of XCQ is that is requires prior schema knowledge (DTD) and can only process XML documents conforming to the given schema. In practice, though, XML documents with unavailable schemas are often used, while in other cases, the document structure can be changed. Another limitation of XCQ is that this approach is clearly tailored to support value-based processing and not structural operations. The only option for providing structural navigation is navigating through DTD edges in a depth-first manner so that the structure of the original document can be re-constructed with the use of the structure stream. This can impact the identification of ancestor-descendant node relationships (not to mention sibling and preceding-following relationships), since large parts of the document that are irrelevant will be accessed. In addition, this approach only supports non-recursive DTDs which limits its applicability by a large factor[3]. The authors claim that it is straightforward to provide support for recursive DTDs. While this is probably true for the compression of an XML document that conforms to a recursive DTD, we believe that during query evaluation, recursion will further add to the structural navigation complexity and query evaluation performance will further degrade. On the contrary, XCQ provides good sup-

[3] An early study conducted on real life DTDs showed that 35 out of 60 DTDs are recursive [29].

port for value-based predicates processing. The partitioning of each data stream into blocks that are independently compressed, provides a partial decomposition approach, while the Block Statistics Signature indexing scheme can efficiently locate the relevant blocks. Due to the proposed Striping decomposition, the textual compression of XCQ (which is independent of the DTD) can be directly applied in any of our storage schemes. Interestingly enough, we proposed a similar approach for the structural compression at the structural agnostic compression storage scheme. To that end, we regard the XCQ textual compression approach as complementary to our approach which targets structural compression.

Another queryable compressor approach is that of *XQueC* [9], which stands for *XQue*ry processor and *C*ompressor. XQueC, as a non-homomorphic compressor, is based on the idea of separating the XML content from the XML structure. XQueC stores the XML structure using a structure tree and an auxiliary structure, the structure summary. In addition, for storing data values, the path-based approach is preferred; data items reached by the same root-to-leaf label-path are grouped into the same value container. The structure tree is stored as a set of node records. Each record represents a non-value document node and is assigned a unique id, a tag code and the ids of its children and parent nodes. If a node has an associated value, then it also points to the value in the respective container. Node records are stored in document order. The structure summary is a redundant structure that stores all unique label-paths in a document and is used as an extra access method for efficient query evaluation. Each summary node stores the ids of the structure tree nodes that correspond to the same label-path. In addition, each leaf-node points to the respective value container. Similar to homomorphic approaches such as XGrind and XPress, XQuec provides a fine-grained data compression approach, *i.e.,* the data items are compressed individually. A value container is a sequence of container records, each of which consists of a compressed data value and a pointer to the parent of this value in the structure tree. Container contents do not follow document order. Instead, these are stored in lexicographical order to enable fast value-based predicate processing. XQueC's unique feature among other XML compressors is the ability to select a compression strategy, based on the data commonalities and an expected query workload. In detail, XQueC selects from a variety of compression algorithms that support different features such as equality or wildcard matching. The appropriate compression algorithm is then decided by exploiting data commonalities according to the predicates present in a given query workload. Thus, the set of data containers may be logically partitioned so that each group can share the

same source model for the selected compression algorithm that will be applied. To select the grouping of containers along with the appropriate compression algorithm, the system employs a cost model and a greedy algorithm. The fine-grained data compression approach in conjunction with the value container grouping and the tailored data compression algorithms, enables the evaluation of value-based predicates directly on the compressed data, resulting in large computational and space savings. In addition, the use of the structure summary can significantly reduce the I/O cost by selecting only the value containers that contain relevant to the query data. XQueC provides a variety of structures that enable various evaluation strategies for the efficient processing of a large fragment of XQuery. However, the structure summary and especially the structure tree along with all the pointers from and to compressed values incur a huge space overhead. In addition, the structures can merely provide parent-child navigation and thus large parts of the XML structure may still be accessed, although not required.

Most of these limitations were tackled by an improved version of XQueC, described in [10]. XQueC's storage model was essentially re-designed and the single-dimensional node identifiers were substituted by the two-dimensional structural identifiers of the PrePost labelling scheme. This allowed node identification in constant time. The storage structures of XQueC were augmented with the new structural encoding. The query engine was also enhanced with efficient structural join operations and XML pattern matching techniques. To efficiently support such operations, the structure tree was decomposed in a path-based approach to provide only relevant document parts as input. To that end, the structure tree was encoded as a set of structural ID sequences ($(pre, post)$ sequences), each associated with a unique document rooted label-path. This is exactly the Striping decomposition for the structural part of an XML document. In addition to the enhanced storage model and query engine, XQueC also enhanced its greedy, cost-based approach of selecting value container groupings and compression algorithms. Local greedy optimisations were proposed for deciding on a good configuration of containment grouping and compression algorithms. The decision is still based on a given query workload and especially their value-based predicate expressions. The experimental results showed that the cost-based approach always provides better compression strategies than the naïve hard-coded alternatives. The achieved compression ratios of XQueC were always close to the ones achieved by other queryable XML compressors such as XGrind and XPress. In addition, a potential compression ratio decrease is balanced by the XQueC's query capabilities which cover a rich subset of XQuery. One of the limitations of XQueC is that the supported XPath

fragment ignores horizontal navigation, *i.e.,* sibling and preceding-following node relationships. In addition, the experiments scale up to a document size of 115MB and thus, they do not provide any insight of the compression impact on really big XML documents. Finally, regarding the comparison of the no-compression version of XQueC with a compression-unaware XQuery processor, the Galax [38] implementation was selected. However, we consider Galax as a reference system for the implementation of XQuery and it is known that there exist many other XQuery implementations that are much more efficient, such as Monet/XQuery [17].

6.7.3 Discussion

Our proposed Striping model is based on the separation of the XML structure from XML data. Thus, the non-homomorphic, queryable compressor approaches are mostly related to our proposed compression storage schemes. The XCQ compressor [79] is a schema-aware compressor and is thus tightly coupled to documents that conform to a specific schema. The XQzip [27] and XQueC (early version) [9] approaches compress the document structure by employing mainly main memory structures that may scale to a prohibit size, especially when really large XML documents are considered. These limitations were handled by the later version of XQueC [10], by storing the document structure in ID sequences, using a path-based partition setup on persistent storage. This is exactly our persistent-storage approach implemented as the explicit storage scheme. In addition, we proposed two compression storage schemes, one for exploiting document structure regularities and another that is immune to them. Our experimental results showed that by compressing the document structure on persistent storage, query evaluation performance is at most times enhanced. On the other hand, all these XML compressors focus on XML data compression and we all share a common ground; the path-based partitioned value containers[4]. Thus, their proposed compression techniques for the XML data part of the documents can be directly applied to any of our storage schemes, although it would be mostly natural to be combined with any of the proposed storage schemes that apply structural compression. To that end, we regard these approaches as complementary to our compression schemes. For XML data compression, there exist two main compression approaches that differ in terms of compression block granularity. The first, adopted by XQzip and XCQ is to further partition a value container into data blocks of a certain size and then compress each data block

[4]XQzip actually proposes label-based partitioned value containers but the same approach could also host path-based partitions.

individually. The second approach, used by the XQueC compressor (but already proposed by the homomorphic queryable compressors) is that of a fine-granularity compression, which operates on each data value individually. Each compression technique has certain advantages and disadvantages. Coarser-granularity compression produces better compression ratios as the compression encodings can exploit repeated values and produce smaller compressed blocks. On the other hand, compressing each data value in isolation may not produce a very compact result, however, with the appropriate selection of compression algorithm, the evaluation of certain predicates can be directly applied on the compressed domain, avoiding completely the decompression cost. It is not clear which is the best approach for compressing data values. However all the proposed techniques can be implemented on top of our Striping model.

Chapter 7

Conclusion

In this chapter, we summarise the proposed decomposition model and present its most important benefits. In Section 7.1, we review the benefits of the proposed model and optimisations. In Section 7.2, we review the proposed storage schemes that can be directly applied under the general decomposition model and compare their performance. In Section 7.3, we discuss the benefits and drawbacks of the proposed model with respect to other proposed techniques and decomposition methods. Finally, in Section 7.4, we summarise the most interesting findings.

7.1 Impact of Striping and Optimisations

We now summarise the impact of Striping and the proposed optimisations. Throughout the discussion we will be referring to concrete findings from our experimental evaluation.

7.1.1 Striping

We proposed a data model for storing and querying XML data, which is based on a general decomposition method. We employed a vertical partitioning technique, Striping, according to which all document nodes are clustered based on their label-path (Stripes). One of the natural benefits of the decomposition that the proposed data model enforces, is that it provides the means to effectively minimise query input, so that the query engine touches only data that are relevant to a given query. This is the outcome of Stripe Projection, the first part of the Input Minimisation process (Section 2.3), which is a direct benefit of Striping.

The biggest impact of Striping in terms of the number of Stripes being reduced occurs for the Xmark dataset. The average selected number of Stripes that need to be accessed for query evaluation, is less than 4% of the total number of Stripes that comprise the Xmark dataset. This is mainly due to the large number of unique label-paths that exist in the Xmark document tree in addition to the fact that the proposed storage model separates structure from content. This results in a large degree of data separation (the total number of Stripes for the Xmark dataset is 1046) which provides the potential of selecting small parts of the dataset that are relevant to a given query. Similar results are observed regarding the cardinality of the selected Stripes as well as their size. The average number of Stripe nodes of the selected Stripes is 3.3% of all nodes of the Xmark dataset, while the average size of the Stripes being selected for an Xmark query, consists of 2% of the total size of all Xmark Stripes.

In contrast to the Xmark dataset, the smallest impact of Striping on input reduction is reported for the Mbench dataset. This is due to the small number of unique label-paths of the Mbench document tree and its highly recursive structure. For most of the Mbench queries, a large part of its structure and/or content is involved. Despite that, the number of the selected Stripes due to Striping comprises 43% of the striped dataset on average, while the Stripe size is merely 10% of the total size of the dataset, as the Stripes that contain long textual data (and are not needed) are effectively pruned.

Finally, for the DBLP dataset, the number of selected Stripes comprises on average 27% of the total number of Stripes, achieving significant Stripe pruning. In addition, the average size of the selected Stripes is 20% of the total size of the striped dataset.

7.1.2 Pruning

We also presented Stripe Pruning (Section 2.3.3), which is the complementary part of Stripe Projection. Stripe Pruning, or simply Pruning, operates on the selected (set of) Stripes as produced from the Stripe Projection process. Based on the overall query semantics, Pruning may further prune Stripes that are guaranteed not to contribute to the query result.

Pruning has minimal impact on the Xmark queries. For all Xmark queries but two, Pruning has absolutely no effect since no Stripe can be further reduced from the set of Stripes initially selected for query evaluation. This is also reflected on query evaluation performance, where the PR evaluation setup performs better than the NV setup only in two queries. The average number of Stripes being further reduced due to Pruning is

merely of 6%. Regarding the size of the Stripes, the average reduction for the Xmark queries is close to 5%.

On the other hand, Pruning has a huge impact on input reduction for the majority of the Mbench queries. As already reported, Stripes are further reduced by 62% on average, while for almost half of the Mbench queries, Stripes are reduced by at least 80%. This is also reflected to the average Stripe size that is reduced by 50%, compared to the size of the selected Stripes when Pruning is not applied. The input reduction is also reflected on query evaluation. The PR evaluation setup performs faster than the NV setup by almost 60% on average.

The impact of Pruning on query input is also significant for the DBLP dataset. The added reduction of Stripes when Pruning is applied is in the order of 48% on average. However, Pruning affects half of the DBLP queries, while for the other half of the queries, Pruning has minimal or small effect. In addition, the reduction of Stripes does not always implies proportional reduction of Stripe size. Regardless, the achieved average size reduction is of 40%. This is faithfully reflected in the response times of the PR evaluation setup, which outperforms the NV setup by 40% on average.

7.1.3 Rewriting

We also presented the Path Minimisation process (Section 2.4), or simply Rewriting, which operates on path expressions and reduces them by applying path equivalence rules that hold over the proposed striped model. The benefits of Rewriting are twofold. It may reduce (a) the number of input Stripes, and (b) the number of both scan and structural join operations needed for the evaluation of the original expression.

For most of the Xmark queries, Rewriting significantly reduces the number of input Stripes by 52% on average. However, as already described, the Stripe reduction is not always reflected in the total Stripe size reduction achieved, which is in the order of 29%. In addition, apart from the Stripe scan operation reduction of 53%, which is a direct effect of Stripe reduction, the structural join operations are also reduced by 82% on average for the Xmark queries. However, the query evaluation results show that query performance is mostly affected by the input size reduction while being almost immune to the operation reduction accomplished by Rewriting. The RW evaluation setup outperforms the NV setup by 27% on average.

On the other hand, Rewriting does not have a similar impact on the Mbench dataset. For most of the Mbench queries, structural selectivity is controlled by value-based

predicate expressions. This results in path expressions for which the equivalence rules were seldomly applied. As a consequence, Stripes are further reduced only by 20%, on average for the Mbench queries, while due to the fact that the pruned Stripes contained a small portion of document nodes, the achieved size reduction is in the order of 5%. The Stripe reduction also led to a Stripe scan operations reduction of the same order. In addition, structural join operations were reduced by 50% on average. Nevertheless, query performance is mostly affected by the minimal impact of Stripe size reduction. For most of the Mbench queries, the evaluation times of the RW evaluation setup are comparable to those of the NV setup. A few only exceptions exist due to a sizeable Stripe size reduction in addition to the reduced number of structural join operators employed.

Finally, for the DBLP dataset, Rewriting has an important effect on the reduction of the query input. For most DBLP queries, it effectively reduces the number of the selected Stripes by a factor of 60% on average. Again, however, the average Stripe size reduction is not always reflected by the average number of Stripes being reduced. The Stripe size is reduced by 40% on average. Nevertheless, the effectiveness of Rewriting on input reduction is satisfactory. The query evaluation results of the RW evaluation setup performs better by a factor of 35% on average, compared to the NV evaluation setup. Rewriting also achieves a significant reduction of scan (40%) and structural join (70%) operations. However, as in the case of the Xmark and Mbench datasets, the query evaluation performance was heavily affected by the size reduction of the query input, which was accomplished by the reduction of input Stripes. The reduction of structural join operations rarely has an impact on the produced results. This effectively shows that the dominant factor for query processing is the I/O cost.

7.1.4 Stripe-Aware Optimisation

We proposed evaluation algorithms and access methods for evaluating queries in $\mathcal{XP}$, a large fragment of XPath. We also explored whether and under which circumstances, these evaluation algorithms can be enhanced due to the proposed decomposition model (Section 3.4). Stripe-aware Optimisation involves the selection of a Stripe-aware algorithm for an operator's implementation, whenever this is possible.

Stripe-aware Optimisation has minimal effect on query performance for the Xmark and DBLP queries. For the Xmark dataset, there is only one exception, for which the Stripe-aware evaluation algorithms result in skipping large portions of the input

Stripes. Overall, however, its effect is rather minimal to query evaluation results. Regarding the DBLP tested queries, the impact of Stripe-aware Optimisation is more evident compared to that of the Xmark queries. The evaluation results for four DBLP queries benefit from applying Stripe-aware algorithms, achieving a 35% improvement in the best case. However, the average improvement is merely in the order of 5%, while in certain cases, the extra complexity of the Stripe-aware algorithms hurts the query evaluation results.

On the other hand, Stripe-aware Optimisation impact is evident for the Mbench dataset, which is characterised by its highly recursive structure. The specialised Stripe-aware algorithms effectively access a minimal set of document nodes; the nodes being accessed from the scan operators are reduced by 80% when the Stripe-aware Optimisation is enabled, since they manage to reduce the I/O cost by retrieving smaller parts of a Stripe. This is verified experimentally by the query evaluation results of the OP evaluation setup, which are improved by 40% on average compared to the NV setup.

7.2 Storage Schemes Comparison

We proposed three alternative storage schemes under the general decomposition model. The schemes differ in the compression method imposed on the structural part of the XML document. The first and most natural storage scheme, the explicit storage scheme, is one where no compression is imposed, and as such, each node in a Stripe corresponds to a single document node. The tree-sharing compression storage scheme exploits structural regularities in the document to minimise storage and, thus, I/O cost during query evaluation. Finally, the agnostic compression storage scheme performs structural-agnostic compression of the document structure which results in minimised storage, regardless the actual XML structure.

We now compare the query evaluation efficiency of the query engines over each of the proposed storage schemes (SRX, CSRX and ASRX). The reported results concern the PRO evaluation setup, *i.e.,* when all the proposed optimisations (Pruning, Rewriting and Stripe-aware Optimisation) are enabled. We also include the results of the state-of-the-art system in XML query processing, MDB.

The evaluation results of all tested systems for the largest of the Xmark datasets (sf=100, size=11GB) are depicted in Figure 7.1. Regarding the tree-sharing compression storage scheme, we have that for 70% of the Xmark queries, CSRX performs worse

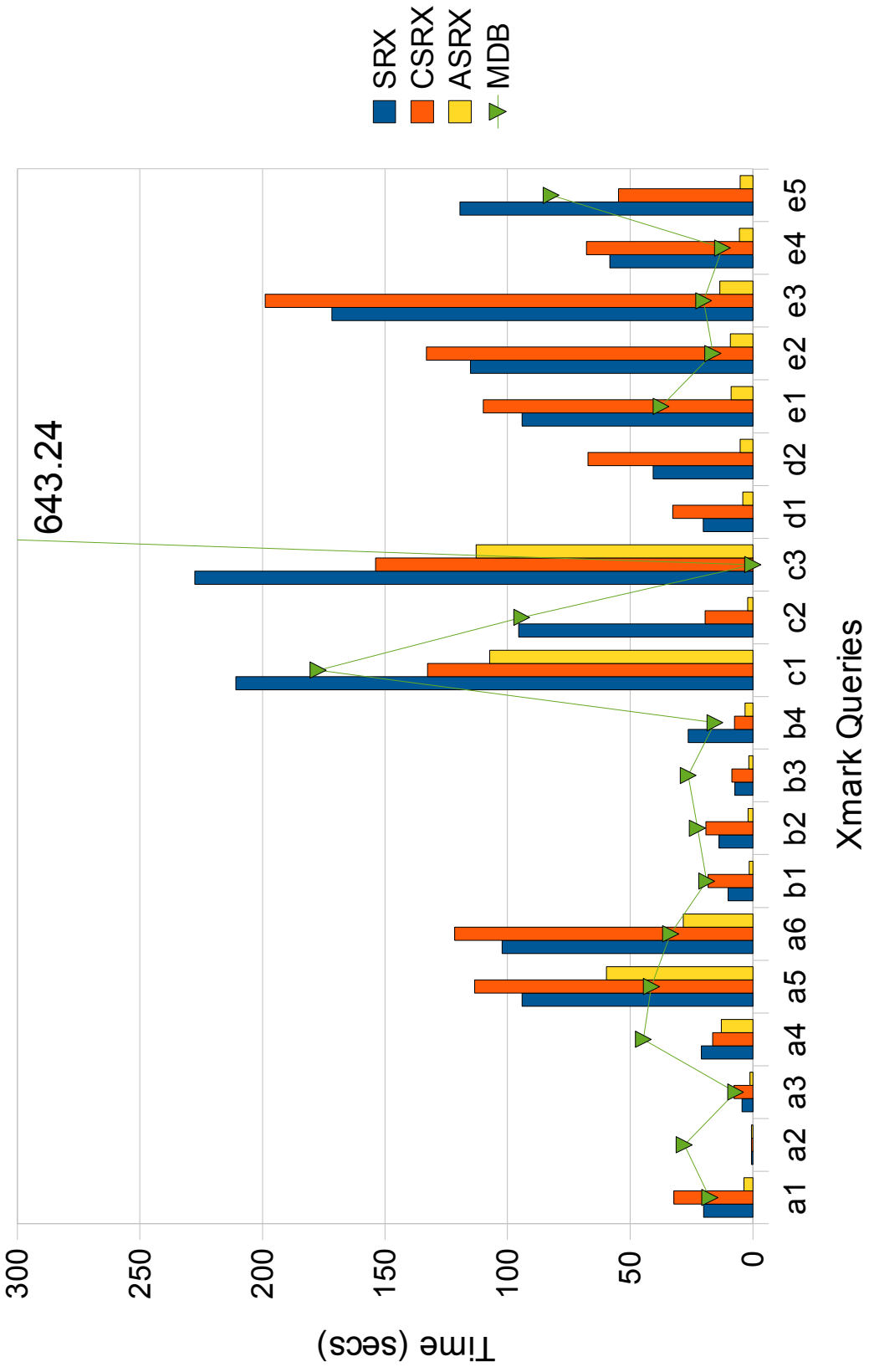


Figure 7.1: Comparison for the Xmark100 dataset

than SRX by 25% on average. This is due to the poor compression of the Xmark dataset, that for those queries results in low input size reduction compared to the input size for SRX. On the other hand, CSRX is the clear winner (by almost 50% on average) for the rest of the Xmark queries (30% of the Xmark query testbed), where compression has a significant impact on the Stripes involved in those queries. As already described, there exist a few Xmark queries (*e.g.,* b2, b3) where the small I/O benefits from the poor compression are covered by the extra computational cost needed for node decompression. Regarding the agnostic compression storage scheme, the compression impact on query evaluation is profound; ASRX outperforms SRX for all Xmark queries (but one) by a large factor of 78%. Even for the query that SRX performs better than ASRX, the difference is insignificant. In addition, ASRX outperforms CSRX for all Xmark queries by 71%. Overall, we conclude that for the Xmark dataset, the explicit storage scheme is more suitable over the tree-sharing compression scheme. Nevertheless, the clear winner among the proposed storage schemes is the one that applies the agnostic Compression. ASRX also outperforms MDB for almost all Xmark queries by a large factor (74%). The only cases where MDB is superior to ASRX are queries a5 and c3, whose evaluation is dominated by very selective value-based predicates. However, when comparing MDB to SRX or CSRX, there is no clear winner. Each of SRX and CSRX outperforms MDB in almost half of the Xmark queries. The detailed comparisons are described in Sections 4.5.4 and 5.6.5.

We proceed to the comparison of all systems for the largest of the Mbench datasets (sf=10, size=5GB). The evaluation results are displayed in Figure 7.2. Regarding the tree-sharing compression storage scheme, we have that CSRX outperforms SRX for all Mbench queries (but one) by a factor of 37%. This is due to the significant compression that effectively reduces the size of the selected for query evaluation Stripes (by 25% on average). Similar results hold for ASRX, which outperforms SRX for all Mbench queries by 40%. The evaluation results for both compressed schemes are very close. In detail, ASRX outperforms CSRX for the 70% of the Mbench queries by a factor of 8%, while on the other hand, it performs worse in the remaining 30% by a factor of 3%. Thus, we conclude that for the Mbench dataset, both compression storage schemes exhibit similar performance in terms of compression effectiveness and query evaluation performance and are thus preferred over the explicit scheme. In addition, the agnostic compression storage scheme performs slightly better than the tree-sharing compression scheme and is thus the best option for the Mbench dataset. Compared to the MDB

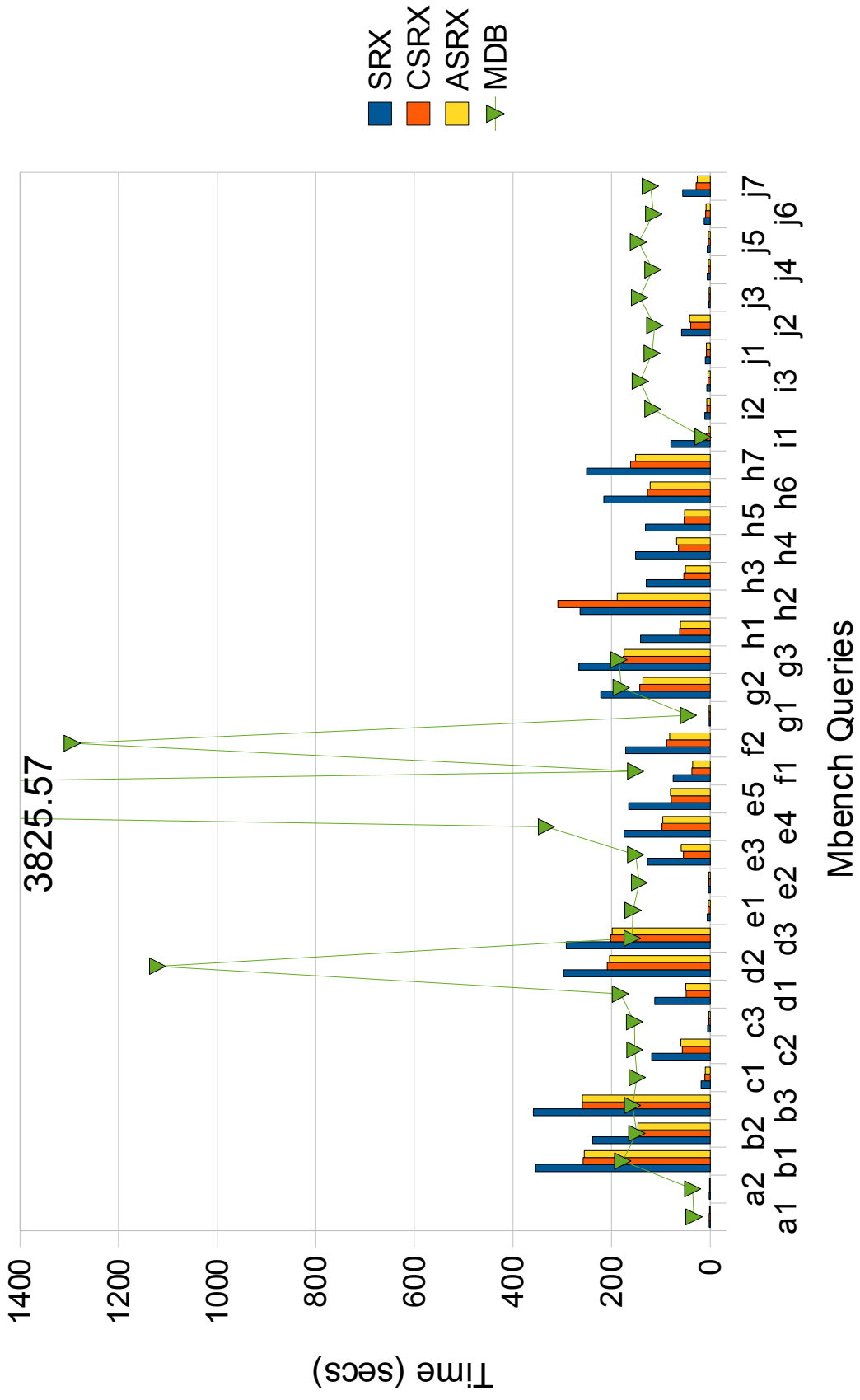


Figure 7.2: Comparison for the Mbench10 dataset

results, SRX, CSRX and ASRX outperform MDB for a large part of the Mbench query testbed. Especially for the two compressed schemes, we have that CSRX outperforms MDB in 89% of the Mbench query testbed by 80% while ASRX outperforms MDB in 92% of the Mbench query testbed by 79%. Again, the cases where MDB is superior to any of CSRX, ASRX are those whose evaluation is dominated by very selective value-based predicates.

Finally, we present the comparison of all systems for the largest of the DBLP datasets (sf=10, size=1GB). The evaluation results are displayed in Figure 7.3. Regarding the tree-sharing compression storage scheme, we have that CSRX outperforms SRX for all DBLP queries by 49% on average, while being, at worst, comparable to SRX. Again, this occurs due to the compression effect on Path Stripes. The size of the selected for query evaluation Stripes is effectively reduced by 80% on average compared to the size of Stripes required by the explicit storage scheme. Regarding the agnostic compression storage scheme, ASRX results are comparable to CSRX. The average Stripe size is reduced by almost 60% on average and this is directly reflected in query evaluation performance; ASRX outperforms SRX for all DBLP queries by 47.5% on average. Compared to CSRX, ASRX performs better than CSRX in 57% of the DBLP queries by a factor of 25%, while CSRX performs better than ASRX in the remaining 43% of the DBLP queries by 11%. As a result, for the DBLP dataset, the explicit storage scheme underperforms compared to the compressed schemes which produce comparable results. Compared to the MDB results, both systems using the compressed schemes outperform MDB in 65% of the DBLP query testbed by almost 90% on average, improving over the performance of SRX. For the remaining DBLP query testbed (35%), MDB continues to perform better by a factor in the order of 50% due to the selective value-based predicate expressions, which none of our proposed schemes evaluates efficiently due to the lack of indexing.

7.3 Discussion

We described a storage model for XML that is reminiscent of vertical partitioning and explored its effectiveness for storing and querying large XML repositories. The most distinctive feature of our storage model is that by decomposing large fragments of the original structure into smaller, path-based fragments, it is easier to select parts of the document that are relevant to a given query. This effectively enables us to minimise the

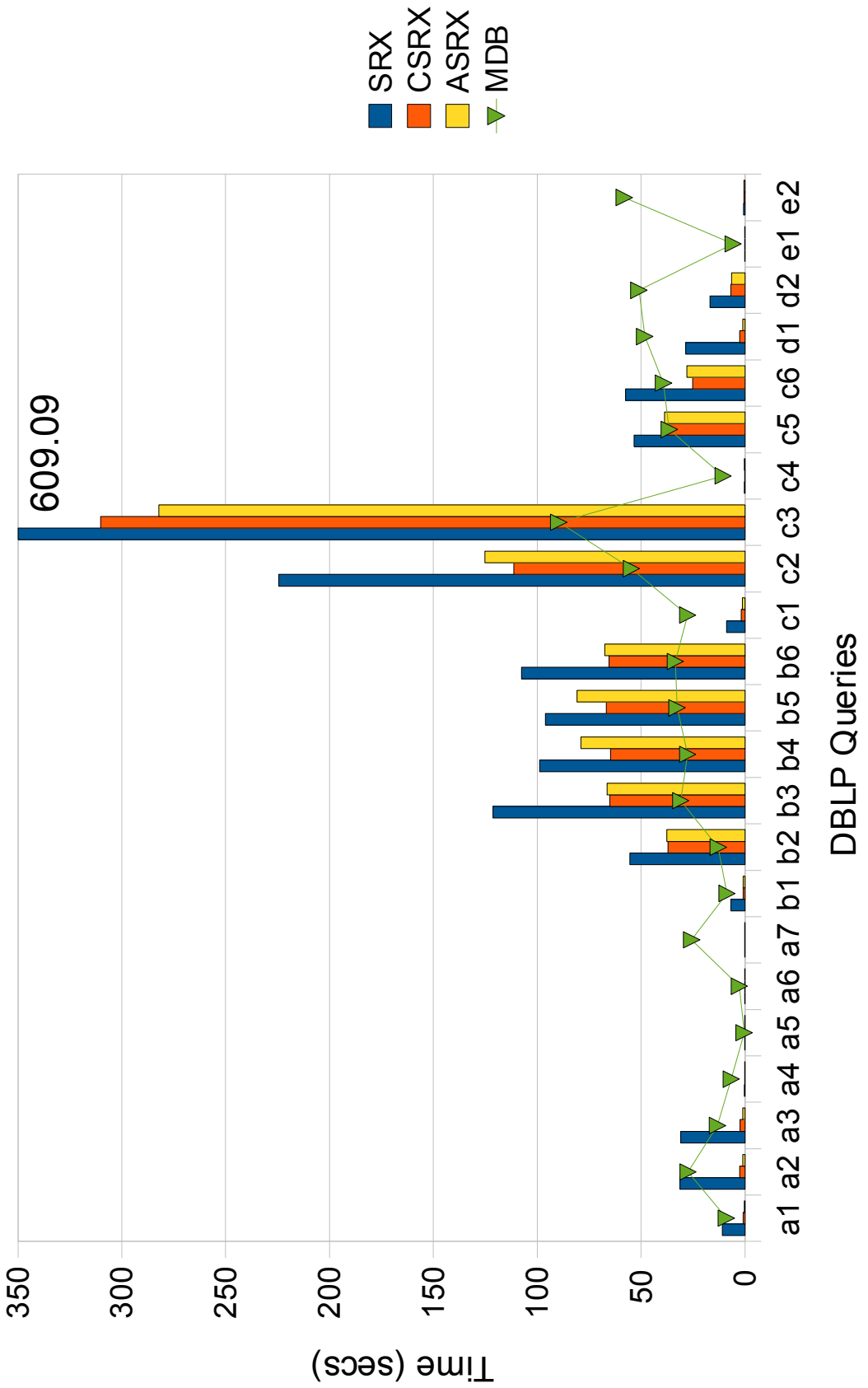


Figure 7.3: Comparison for the DBLP10 dataset

I/O cost and thus the total evaluation cost of a query. On top of that, and in addition to other approaches that applied similar decomposition methods for managing XML data (*e.g.,* [87, 85]), we applied structural-join based evaluation techniques and focused on further optimisation opportunities that arise due to the new representation. To that direction, we described:

- Stripe Pruning, for pruning Stripes based on the overall query semantics that are guaranteed not to contribute to the query result.
- Path Minimisation, for reducing path expressions by applying path equivalence rules that hold over the proposed model, and
- Stripe-Aware Optimisation, for providing evaluation algorithms tailored to the proposed model.

Furthermore, we explored three storage schemes under our general decomposition technique, which differ in the compression method imposed on the structural part of the XML document.

What seems to be the most natural benefit of Striping, is at the same time a potential drawback. Due to the large fragmentation of XML nodes (based on their label-path), it is possible, subject to a given query, to perform excessive merging of a large number of input Stripes in order to provide all necessary input data. This is mostly evident at the serialisation phase where the tree structure of the result nodes must be recreated (if necessary). The serialisation of a selected XML node having a large subtree will possibly involve a large number of Stripes (matching all possible paths for any node of its subtree).

Another drawback of this work is the lack of content indexing. As already described, our work focuses on the evaluation of the structural part of XML queries, while it treats content-based predicates as post-filters. In many cases content-based predicates can be more selective than structural predicates and thus it is necessary that both kinds of predicates are handled efficiently. To efficiently process content search, the database community has adopted methods that were originally used in information retrieval for performing text search. To that extend, many of the already proposed native XML stores (*e.g.,* [55]) are equipped with inverted list structures that allow text searching efficiently using text terms. These techniques have been extensively used, they perform well and they can be directly applied in our proposed decomposition. This is exactly the reason that we turned our focus on the structural part of XML queries

and as shown, the proposed decomposition, combined with structural compression can have a big impact on query evaluation performance. Content-based predicate evaluation techniques can be directly incorporated in our system to efficiently support the selection of all text nodes that contain a certain term. Furthermore, these techniques can be enhanced to fit our proposed decomposition. For instance, if the search terms are also enhanced with label-path information then this extra information which can be deduced at compilation time while performing Stripe selection and pruning, can further assist the text search, narrowing down the index results.

As already described, the main advantage of Striping is that it provides the means to efficiently identify the parts of a documents that are relevant to a given query. As such, the large degree of fragmentation that Striping imposes, serves this purpose well. However, when non-static repositories are considered, the cost of Stripe maintenance can get significantly large. In general, an update operation involves two distinct steps. During the first step, the requested data that is to be updated, is identified while at the second step, the update operation is applied. As a result, an update operation encapsulates a query operation. And while we have demonstrated the effectiveness of of Striping at query evaluation due to the large degree of data fragmentation, this is exactly the reason that for the underlying storage model, XML updates come at a cost.

We now review some of the issues that emerge from the proposed decomposition. XML updates typically fall within two categories. When the update operation merely involves a change of an element or attribute value *i.e.,* leaves the structure of the XML tree intact, then these are characterised as *content updates*. Content updates are easy to handle in our proposed decomposition. Since the text nodes that are to be updated have been located, we simply replace their corresponding values with the new values, at the designated Value or Attribute Stripe.

On the other hand, when the update operation mainly affects the data structure, *i.e.,* involves node insertions, removals and/or label renaming, then these are termed *structural updates*. We now discuss various cases of XML structural updates.

Node deletion. When a target node is selected for deletion, then all nodes that are reachable by that, must be also removed. Thus, a node removal operation will involve as many node deletions as the size of the target node's subtree. In our representation, this means that a delete node operation may involve a large number of (Path, Attribute and Value) Stripes, which can be, at the worst case, equal to the size of the target node's subtree, if each node belongs to a distinct Stripe. The set of Stripes that will

be involved to the update operation can be derived from then system catalog using the label-path of the target node. For each of the Stripes that may contain a node in the target node's subtree, all nodes that lie within its interval (*i.e.,* [start,end)) must be removed.

Node insertion. A node insertion is an update operation that inserts a node (along with its subtree), to a position that is specified by a target node, as well as a placement argument. Thus, it can be specified that the new node(s) will be inserted before or after the target node (as siblings) or as a child (first, last) of the target node. As a consequence, the set of Stripes that will be involved in the update operation is strongly coupled to the target nodes as well as the placement argument with respect to the target node. As soon as the target node is identified, then proper structural information must be allocated to the new node(s). Again, this is subject to the placement argument. For instance, if the new node must be inserted after the target node, then for the new node the following must hold: (a) its start value must be greater than the end value of the target node, and (b) its end value must not exceed the start value of the current target node's following sibling node. If any of these condition cannot bet met, then a re-organisation of a possibly large part of the document is necessary in order to accommodate the update operation. Due to Striping and the large degree of fragmentation imposed by it, this is a very expensive operation and thus extra attention must be given at the initial allocation of structural information on XML nodes during the initial document serialisation to Stripes. As in the case of node deletion, an insert operation may involve a large number of Stripes, when a subtree is to be inserted. Again, these are identified using the system catalog and the target node's label-path.

Node replacement. A node replacement operation is a combination of a node removal and a node insertion operation in the position of the root of the subtree that was just removed. A prerequisite for performing a replace operation without reordering, is that the new subtree/node size must be equal or smaller compared to the size of the subtree/node that is to be deleted. Otherwise, proper re-organisation must be first performed. Note that the sets of Stripes that are involved during the removal and insertion operations can be disjoint. For this reason and subject to the subtrees that are removed/inserted, a replace operation may possible involve a large number of Stripes.

Node renaming. Node renaming usually involves a single node. However, due to Striping and the fact that its label value changes and so does the label values of all of its descendants, a rename operation may result in an expensive update operation.

To handle node renaming, the target node along with its descendant nodes must be removed from their original Stripes and be inserted into different Stripes based on their corresponding, new label-path information. Thus, although a rename operation may involve a single XML node, the whole subtree is affected.

When any of the compressed Storage Schemes is used then the cost of the update operations is also burdened by the extra decompression and re-compression cost of the nodes being updated. For instance, in the case of the Agnostic compression Storage Scheme, a node insertion will result in decompressing at least one shared node in all Stripes that are involved in the update operation and then re-compressing them when the new nodes are inserted. In addition, if the newly inserted nodes cause a shared node to exceed the size of an disk page, then the shared node must be split in two new shared nodes. Similar notes apply for the Tree Sharing Storage Scheme, only that due to the fact that the node sharing occurs in a bottom-up manner and based on the structure of subtrees, update operations may result in splitting shared nodes in many of the Stripes along the label-path of the target node. This effectively makes the Tree Sharing Storage Scheme impractical for non-static XML repositories since the maintenance cost can be prohibit.

Apart from the Striping decomposition model (path partitioning), a few other partitioning schemes and techniques have been proposed over the last few years for supporting the XML data model.

To enable XML support in relational databases, a fixed mapping from the XML structure to a collection of relational tables in the database schema is required. The process of converting XML data to a relational format is known as XML shredding, and it was well studied since the early years that XML received attention [91, 15]. In [91] a set of simplifications is applied on the source XML schema to produce the final schema that can be easily mapped on a set of relations, while in [15], similar transformations are applied over the source schema but only to produce alternative mappings. The best of the alternative mappings is then selected based on schema statistics and a given query workload. In both proposals, XML data is flatted into a set of relations and the hierarchical relationship among XML elements is mostly handled by join operations. As a result, the trend in both works was to cluster an element together with as many of its sub-elements as possible, in order to reduce data fragmentation and thus the number of join operations needed to reconstruct the hierarchy. Nevertheless, due to the diversity of the XML and relational model, in most real-world applications this naturally leads in data redundancy as well as in large and complex relational target schemas. As a result,

for querying shredded data or reassembling the original documents, complex multi-way joins are still in need. Such limitations were soon understood by all major DBMS vendors, (IBM, Oracle and Microsoft) that shifted away from shredding XML data to the purely relational format. Instead, by using a hybrid model and native XML data types that capture the hierarchical nature of XML data in order to provide better support for XML documents. Overall, XML shredding is a useful technique that provides adequate support to fairly simple, shallow and tuple-oriented XML documents (such as those published from a relational database). However, it can be very inefficient when more complex (in structure) and deeply nested documents are considered since these are difficult to map to a relational schema. On the other hand, the Stripe-based approach provides good support even for complex XML documents due to the path-based decomposition that enables query Input Minimisation as well as Path Minimisation.

A different approach to support XML data is to depart from the relational format and directly reflect the structure and properties of the hierarchical XML model. This approach has been adopted by the native XML systems. Some of them perform structural navigation using structural summaries (*e.g.,* [45, 20]) . Others, although they store XML data at the physical layer in its natural tree structure, they employ structure indexes for efficiently supporting node relationships and thus perform navigation in such a manner (*e.g.,* [63, 55, 39, 51]). These indexes usually operate on the element tag name (label), imitating a tag-based partitioning of XML data (tag partitioning). Another partitioning scheme also proposed was to separate XML nodes based on their tag name combined with their level value (tag+level partitioning) [25]. These three partitioning schemes (tag, tag+level and path) differ in the degree of XML node fragmentation they impose. Tag partitioning is the least fragmented of all three; there exist as many node partitions as the number of unique tag names in an XML document. Path partitioning (Striping), on the other hand, is the most fragmented scheme; the number of partitions equals to the number of unique rooted label paths. Tag+level partitioning lies between the two extremes producing a partition for each unique tag name per tree level. Among the three partitioning schemes, the path-based scheme provides more optimisation opportunities with respect to the other two alternatives. This is due to its large degree of fragmentation combined with the hard-encoding of label paths and the nature of the XML tree-model itself. By using Input and Path Minimisation techniques, it is possible to restrict the query input to the partitions that contain data relevant to a given query. Such techniques, although they can be applied to any of the other two partitioning schemes, it is unlikely to have the same effectiveness as in the path parti-

tioning scheme. This is of course subject to the "shape" of the target XML document as well as the query itself. On the other hand, when the Input and Path Minimisation has no effect and the query input is the same for all three schemes, then the path partitioning scheme may underperform compared to the other alternative schemes due to XML node merging.

An interesting direction of future work is to be able to identify the best mapping for an XML document based on a query workload. Of course, prior knowledge of the XML schema is necessary in that case.

In general, the proposed XML store was designed to efficiently store and query large XML repositories that are schema independent. Under this constraint, we proposed a fixed mapping for storing XML data and explicitly stored schema information as metadata in the store catalogs. Part of query processing is delegated to metadata (catalog) processing during query compilation, where by accessing appropriate schema information we optimise query processing whenever possible. For instance, by capturing Stripe (label path) dependencies, we manage to prune Stripes that do not participate in query evaluation and thus skip parts of the document that are irrelevant to a given query (Input Minimisation). In addition, by using schema information, we perform query transformation that results in smaller path expressions (Path Minimisation) which in turn reduces the number of accessed Stripes as well as the number of structural join operations needed for the evaluation of the original query. In general, catalog processing combined with the path-based partitioning of the XML data, may effectively resolve path constraints, redundant path expressions and is also able to check path satisfiability, as far as the structural part of a given query is concerned.

However, our native XML store could benefit from the presence of an XML Schema. First of all, an XML schema, as opposed to DTDs, can provide type information. Prior knowledge of type information may significantly optimise content storage as well as enable faster data retrieval. In addition, during query evaluation unnecessary casting operations can be avoided. Currently, our prototype stores all XML element content and attribute values as plain text. Another benefit from the existence of an XML schema or DTD is that much of the information that is stored as metadata in our catalogs could be deduced by operating in a small, memory-resident schema graph which can be produced by the schema information. In many cases it would be much cheaper to traverse parts of a small DTD graph in main memory that probing B^+-tree structures for catalog retrieval. A possible drawback in this case, however, is when the XML schema is more generic than the actual XML document that conforms to that schema.

In that case, certain optimisations would not be possible to be applied since they would require exact knowledge of the XML structure which can be retrieved by accessing the catalog.

We conclude with a few words regarding the document shredding process. In an attempt to support query processing for any XML document regardless whether it is accompanied with its corresponding schema or not, we built our native XML store based on a fixed, schema-oblivious mapping: XML data is partitioned according to its root-to-leaf label path. This is in contrast to other, schema-aware mapping proposals where a target schema is selected from many alternative mappings based on the source schema information and/or a given query workload(*e.g.,* [91, 15]). Due to the proposed decomposition, there exists an one-to-one correspondence between the source and the target "schema": the target schema always contains as many Stripes as the number of unique label-paths. During document import, XML element, attribute and text nodes are assigned proper structural information and then shredded into Stripes based on their label path.

We have already presented the document shredding process where an input XML document is shredded into a new XML repository. However, sometimes users do not want to coredump the entire document in a database but, instead, they want to select part of the data and store them in an existing database. In general, since our schema mapping is fixed, any generic method that supports selective XML storage can be also applied to our store.

There exist various ways to support selective XML storage. The only prerequisite is to define a way to select the specific parts of an input document that are of interest, based on certain predicates. A natural way for selecting document parts that satisfy a set of given predicates is through an XML query, an XPath expression for instance. The most simple, yet naïve way to support selective storage is to import the whole document and then remove the parts that are of no interest, *i.e.,* all nodes that are neither selected by the given XPath query nor belong to their subtrees. The obvious drawback of this method is that not only the unwanted parts of the document are inserted in the target database but also an extra, removal operation is needed. However, since this extra cost incurs only once *i.e.,* when the new document is inserted into the database, in many cases this solution can be acceptable, especially when small parts of the input document are filtered out.

A better method to support selective storage is to be able to identify the requested parts of the document before the actual import and then import only those parts. Since

the selection of the document parts can be expressed as an XPath query, then a possible solution is to use an XPath evaluator over XML streams to select the document parts that satisfy the given predicates on-the-fly and then shred the results into the target database.

Another approach for supporting selective XML storage is described in [36], only that the focus is on existing, predefined target relational schema. In this work, the predicates for input data selection are defined by extending the source XML schema and associating element types with semantic attributes and rules. These rules are executed during the document import and a set of SQL insert statements is produced for populating the existing database with the qualifying XML data. Our decomposition also results in a fixed schema in which each Stripe can be considered as a relation for shredding all XML data of a specific label-path. Thus, similar semantic attributes and rules can be defined on the source XML schema, so that given the label-path at any given time, the updates of the corresponding Stripe of an existing database are produced. The obvious advantage of this approach is its flexibility; it is a general method that can be adapted to any target schema. Nevertheless, in order to do so it requires a source XML schema.

7.4 Concluding Remarks

We summarise our findings in the following:

- The striped data model provides the infrastructure for efficiently selecting the parts of the document that are relevant to a given query. Our experimental results show that for all tested datasets and queries, a large part of the dataset that is not useful for query evaluation is effectively discarded. Striping achieves significant reduction of the query input.

- Pruning, which operates considering the overall query semantics, may further reduce the query input. For the Xmark dataset, due to the large degree of data separation, the minimal set of Stripes needed for query evaluation is already selected from Striping. Thus, for most of the Xmark queries, the impact of Pruning is minimal. For the Mbench and DBLP queries, however, Pruning manages to further reduce the number of the selected input Stripes (and thus the overall query input size), by a large factor.

- Rewriting applies path equivalence rules that hold over the proposed data model and as a result, it may reduce (a) the number of input Stripes, and (b) the number of both scan and structural join operations needed for the evaluation of the original expression. The impact of Rewriting is significant for most of the Xmark and DBLP queries tested. On the other hand, Rewriting has a minimal effect on most of the Mbench queries due to the fact that most location step expressions were also accompanied by predicate expressions; this prohibits the application of the proposed equivalence rules. In any case, the query evaluation results are primarily influenced by the I/O cost reduction due to the input reduction and secondarily by the CPU cost reduction which is caused from discarding redundant structural join operators.

- The benefits from applying Stripe-aware Optimisation, largely vary. Certain Stripe-aware algorithms merely provide better memory utilisation compared to their Stripe-unaware counterparts. This is unlikely to be reflected in query evaluation times. Other Stripe-aware algorithms, try to skip parts of the document that are not needed, at the expense of performing extra computation. Thus, if large parts of the query input are skipped, the I/O cost of the query is reduced and this is reflected in query evaluation times. For most of the Xmark and DBLP queries, where the Stripe-aware algorithms have minimal impact on the query I/O cost, query evaluation results are unaffected, or even worse, negatively affected. For the majority of the Mbench queries though, for which the dataset's highly recursive structure must be processed, the benefits of Stripe-aware algorithms are profound. These algorithms, in addition to the filter expression processing effectively access small parts of the input Stripes and thus reduce the I/O cost by a large factor.

- The compression effect of the tree-sharing compression storage scheme significantly varies according to the document tree's regularity. This is directly reflected on query evaluation performance. For the cases that the compression has no or small effect to the query input size, query evaluation results are burdened with the added decompression cost as well as other implementation-related costs. However, when the achieved compression has a big impact on query input size, this is always reflected in the response times since the I/O cost is reduced by a large factor.

- The compression effect of the agnostic compression storage scheme is immune

to any of the document's structural properties. For all tested datasets, the compression effect on both the cardinality and size of the Path Stripes is profound. This is also reflected in query evaluation performance, regardless of the tested dataset. The evaluation results (except from distinct cases) are better than those produced from both explicit and tree-sharing compression storage schemes. The agnostic compression storage scheme is always the best choice regardless of the dataset's structural properties.

- The query engine over the agnostic compression storage scheme, ASRX, also performs better than MDB, in most cases. In general, the cases where MDB outperforms ASRX, are the cases where very selective value-based predicates dominate the overall query execution time. In the vast majority of the remaining queries tested against all datasets, ASRX outperforms MDB. The same holds for CSRX but only for the Mbench and DBLP datasets, for which it achieves a significant query input size reduction, due to its compression effect. For the Xmark dataset, on the other hand, which is poorly compressed, MDB outperforms CSRX in most cases.

Appendix A

Path Minimisation Equivalence Rules

For all the proofs we provided below, the following hold: All equivalences are written in the form: $q \overset{\text{srx}}{\equiv} q'$, where q and q' are path expressions that belong to fragments $Q : path/step_1/step_2$ and $Q' : path/step_2$ respectively, as defined in Section 2.4. The equivalences are divided based on the axes specification of subexpression $step_1$ of path expression $q \in Q$. We provide the proofs for expressions that their subexpressions of type $step_2$ are simple, predicate-free location steps ($step_1$). It is straightforward to extend our proofs to cover the presence of predicates. To that end, expressions in Q are of the form: $path/a_1{::}n/a_2{::}m$ while expressions in Q' are of the form: $path/a_3{::}m$, where n and m are nodetests, while a_1, a_2 and a_3 are $\mathcal{XP}$ Axis.

For path expression $q \in Q$, we define the following: Let $\mathcal{SS}_{path}$, $\mathcal{SS}_n$ and $\mathcal{SS}_m$ be the StripeSets projected for path expression $path$ and for each of the location steps $step_1$, $step_2$ that follow in q. It holds that: $\mathcal{RS}_q = \mathcal{SS}_m$. From Equation (2.1) we have that for any context node x of any document:

$$\mathcal{S}[\![path]\!]x \subseteq \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \tag{A.1}$$

and thus:

$$y \in \mathcal{S}[\![path]\!]x \Rightarrow y \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \tag{A.2}$$

As already shown in Section 2.4, to prove that $q \overset{\text{srx}}{\equiv} q'$, it is sufficient to show that:

$$\mathcal{F}[\![q']\!](x, \mathcal{RS}_q) = \mathcal{S}[\![q]\!]x$$

A.1 Child Axis

According to the Stripe projection for *child* axis, we have that for StripeSet $\mathcal{SS}_n$ the following holds:

$$\begin{aligned}\mathsf{ss\_nodes}(\mathcal{SS}_n) = \{\, y \,|\, & x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge \\ & \mathsf{path}(x) = \mathsf{maximal\_prefix}(\mathsf{path}(y)) \wedge \tau(y,n) \,\} \\ = \{\, y \,|\, & x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge y \in \mathsf{children}(x) \wedge \tau(y,n) \,\}\end{aligned} \tag{A.3}$$

The following properties are deduced directly from the relationship among nodes:

$$x_2 \in \mathsf{children}(x_1) \wedge x_3 \in \mathsf{self}(x_2) \Rightarrow x_3 \in \mathsf{children}(x_1) \tag{A.4}$$

$$x_2 \in \mathsf{children}(x_1) \wedge x_3 \in \mathsf{children}(x_2) \Rightarrow x_3 \in \mathsf{children}^+(x_1) \tag{A.5}$$

$$x_2 \in \mathsf{children}(x_1) \wedge x_3 \in \mathsf{children}^+(x_2) \Rightarrow x_3 \in \mathsf{children}^+(x_1) \tag{A.6}$$

$$x_2 \in \mathsf{children}(x_1) \wedge x_3 \in \mathsf{children}^*(x_2) \Rightarrow x_3 \in \mathsf{children}^+(x_1) \tag{A.7}$$

Equivalence 2.9: For any path expression *path*, the following holds:

$$path/child::n/self::m \overset{\mathsf{srx}}{\equiv} path/child::m$$

Proof: According to the Stripe projection for *self* axis, we have that for StripeSet $\mathcal{SS}_m$ the following holds:

$$\mathsf{ss\_nodes}(\mathcal{SS}_m) = \{\, x \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge \tau(x,m) \,\} \tag{A.8}$$

From Equations (A.3) , (A.8) we have that:

$$\begin{aligned}\forall z \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \exists x,y : x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \\ \wedge\, y \in \mathsf{children}(x) \wedge \tau(y,n) \wedge z = y \wedge \tau(z,m)\end{aligned} \tag{A.9}$$

Now, for any context node x, we have:

$$\begin{aligned}
\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/child::m]\!](x,\mathcal{RS}_q)\\
&= \mathcal{V}(\mathcal{S}[\![path/child::m]\!]x,\mathcal{RS}_q)\\
&= \{x_3\,|\,x_3\in\mathcal{S}[\![path/child::m]\!]x\wedge x_3\in \mathsf{ss_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3\,|\,x_1\in\mathcal{S}[\![path]\!]x\wedge x_3\in\mathcal{S}[\![child::m]\!]x_1\wedge x_3\in \mathsf{ss_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3\,|\,x_1\in\mathcal{S}[\![path]\!]x\wedge x_3\in \mathsf{children}(x_1)\wedge\tau(x_3,m)\wedge\\
&\qquad x_3\in\mathsf{ss_nodes}(\mathcal{SS}_m)\}\\
&\overset{(A.9)}{=} \{x_3\,|\,x_1\in\mathcal{S}[\![path]\!]x\wedge x_3\in \mathsf{children}(x_1)\wedge\tau(x_3,m)\wedge\\
&\qquad x_1\in\mathsf{ss_nodes}(\mathcal{SS}_{path})\wedge x_2\in\mathsf{children}(x_1)\wedge\tau(x_2,n)\wedge\\
&\qquad x_3=x_2\wedge\tau(x_3,m)\}\\
&\overset{(A.4)}{=} \{x_3\,|\,x_1\in\mathcal{S}[\![path]\!]x\wedge x_2\in \mathsf{children}(x_1)\wedge\tau(x_2,n)\wedge\\
&\qquad x_3=x_2\wedge\tau(x_3,m)\}\\
&= \{x_3\,|\,x_1\in\mathcal{S}[\![path]\!]x\wedge x_2\in\mathcal{S}[\![child::n]\!]x_1\wedge x_3\in\mathcal{S}[\![self::m]\!]x_2\}\\
&= \mathcal{S}[\![path/child::n/self::m]\!]x
\end{aligned}$$

□

Equivalence 2.10: For any path expression *path*, the following holds:

$$path/child::n/child::m \overset{\mathsf{srx}}{\equiv} path/descendant::m$$

Proof: According to the Stripe projection for *child* axis, we have that for StripeSet $\mathcal{SS}_m$ the following holds:

$$\begin{aligned}
\mathsf{ss_nodes}(\mathcal{SS}_m) &= \{y\,|\,x\in\mathsf{ss_nodes}(\mathcal{SS}_n)\wedge\\
&\qquad \mathsf{path}(x)=\mathsf{maximal_prefix}(\mathsf{path}(y))\wedge\tau(y,m)\}\\
&= \{y\,|\,x\in\mathsf{ss_nodes}(\mathcal{SS}_n)\wedge y\in\mathsf{children}(x)\wedge\tau(y,m)\}
\end{aligned}\tag{A.10}$$

From Equations (A.3) , (A.10) we have that:

$$\begin{aligned}
\forall z\in\mathsf{ss_nodes}(\mathcal{SS}_m)\exists x,y: x\in\mathsf{ss_nodes}(\mathcal{SS}_{path})\\
\wedge y\in\mathsf{children}(x)\wedge\tau(y,n)\wedge z\in\mathsf{children}(y)\wedge\tau(z,m)
\end{aligned}\tag{A.11}$$

Now, for any context node x, we have:

$$\begin{aligned}
\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/descendant::m]\!](x,\mathcal{RS}_q)\\
&= \mathcal{V}(\mathcal{S}[\![path/descendant::m]\!]x,\mathcal{RS}_q)\\
&= \{x_3 \,|\, x_3 \in \mathcal{S}[\![path/descendant::m]\!]x \wedge x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![descendant::m]\!]x_1 \wedge\\
&\quad\; x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{children}^+(x_1) \wedge \tau(x_3,m) \wedge\\
&\quad\; x_3 \in \mathsf{ss_nodes}(\mathcal{SS}_m)\}\\
&\overset{(A.11)}{=} \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{children}^+(x_1) \wedge \tau(x_3,m) \wedge\\
&\quad\; x_1 \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}(x_1) \wedge \tau(x_2,n) \wedge\\
&\quad\; x_3 \in \mathsf{children}(x_2) \wedge \tau(x_3,m)\}\\
&\overset{(A.5)}{=} \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{children}(x_1) \wedge \tau(x_2,n) \wedge\\
&\quad\; x_3 \in \mathsf{children}(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![child::n]\!]x_1 \wedge x_3 \in \mathcal{S}[\![child::m]\!]x_2\}\\
&= \mathcal{S}[\![path/child::n/child::m]\!]x
\end{aligned}$$

□

Equivalence 2.11: For any path expression $path$, the following holds:

$$path/child::n/descendant::m \overset{\mathsf{srx}}{\equiv} path/descendant::m$$

Proof: According to the Stripe projection for *descendant* axis, we have that for StripeSet $\mathcal{SS}_m$ the following holds:

$$\begin{aligned}
\mathsf{ss_nodes}(\mathcal{SS}_m) &= \{y \,|\, x \in \mathsf{ss_nodes}(\mathcal{SS}_n) \wedge\\
&\qquad \mathsf{path}(x) \in \mathsf{proper_prefix}(\mathsf{path}(y)) \wedge \tau(y,m)\} \qquad (A.12)\\
&= \{y \,|\, x \in \mathsf{ss_nodes}(\mathcal{SS}_n) \wedge y \in \mathsf{children}^+(x) \wedge \tau(y,m)\}
\end{aligned}$$

From Equations (A.3) , (A.12) we have that:

$$\begin{aligned}
\forall z \in \mathsf{ss_nodes}(\mathcal{SS}_m) \exists x,y : x \in \mathsf{ss_nodes}(\mathcal{SS}_{path})\\
\wedge y \in \mathsf{children}(x) \wedge \tau(y,n) \wedge z \in \mathsf{children}^+(y) \wedge \tau(z,m) \quad (A.13)
\end{aligned}$$

Now, for any context node x, we have:

$$\begin{aligned}
\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/descendant{::}m]\!](x,\mathcal{RS}_q)\\
&= \mathcal{V}(\mathcal{S}[\![path/descendant{::}m]\!]x,\mathcal{RS}_q)\\
&= \{x_3 \,|\, x_3 \in \mathcal{S}[\![path/descendant{::}m]\!]x \wedge x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![descendant{::}m]\!]x_1 \wedge\\
&\qquad x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q)\}\\
&\overset{(A.13)}{=} \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{children}^+(x_1) \wedge \tau(x_3,m) \wedge\\
&\qquad x_3 \in \mathsf{ss_nodes}(\mathcal{SS}_m)\}\\
&\overset{(A.6)}{=} \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{children}^+(x_1) \wedge \tau(x_3,m) \wedge\\
&\qquad x_1 \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}(x_1) \wedge \tau(x_2,n) \wedge\\
&\qquad x_3 \in \mathsf{children}^+(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{children}(x_1) \wedge \tau(x_2,n) \wedge\\
&\qquad x_3 \in \mathsf{children}^+(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![child{::}n]\!]x_1 \wedge\\
&\qquad x_3 \in \mathcal{S}[\![descendant{::}m]\!]x_2\}\\
&= \mathcal{S}[\![path/child{::}n/descendant{::}m]\!]x
\end{aligned}$$

□

Equivalence 2.12: For any path expression $path$, the following holds:

$$path/child{::}n/descendant\text{-}or\text{-}self{::}m \overset{\mathsf{srx}}{\equiv} path/descendant{::}m$$

Proof: According to the Stripe projection for *descendant-or-self* axis, we have that for StripeSet $\mathcal{SS}_m$ the following holds:

$$\begin{aligned}
\mathsf{ss_nodes}(\mathcal{SS}_m) &= \{y \,|\, x \in \mathsf{ss_nodes}(\mathcal{SS}_n) \wedge \mathsf{path}(x) \in \mathsf{prefix}(\mathsf{path}(y)) \wedge \tau(y,m)\}\\
&= \{y \,|\, x \in \mathsf{ss_nodes}(\mathcal{SS}_n) \wedge y \in \mathsf{children}^*(x) \wedge \tau(y,m)\} \qquad (A.14)
\end{aligned}$$

From Equations (A.3) , (A.14) we have that:

$$\begin{aligned}
\forall z \in \mathsf{ss_nodes}(\mathcal{SS}_m) \exists x,y : x \in \mathsf{ss_nodes}(\mathcal{SS}_{path})\\
\wedge y \in \mathsf{children}(x) \wedge \tau(y,n) \wedge z \in \mathsf{children}^*(y) \wedge \tau(z,m) \quad (A.15)
\end{aligned}$$

Now, for any context node x, we have:

$$\begin{aligned}
\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/descendant::m]\!](x,\mathcal{RS}_q)\\
&= \mathcal{V}(\mathcal{S}[\![path/descendant::m]\!]x,\mathcal{RS}_q)\\
&= \{x_3 \mid x_3 \in \mathcal{S}[\![path/descendant::m]\!]x \wedge x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![descendant::m]\!]x_1 \wedge\\
&\qquad x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q)\}\\
&\overset{(A.15)}{=} \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{children}^+(x_1) \wedge \tau(x_3,m) \wedge\\
&\qquad x_3 \in \mathsf{ss_nodes}(\mathcal{SS}_m)\}\\
&\overset{(A.7)}{=} \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{children}^+(x_1) \wedge \tau(x_3,m) \wedge\\
&\qquad x_1 \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}(x_1) \wedge \tau(x_2,n) \wedge\\
&\qquad x_3 \in \mathsf{children}^*(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{children}(x_1) \wedge \tau(x_2,n) \wedge\\
&\qquad x_3 \in \mathsf{children}^*(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![child::n]\!]x_1 \wedge\\
&\qquad x_3 \in \mathcal{S}[\![descendant\text{-}or\text{-}self::m]\!]x_2\}\\
&= \mathcal{S}[\![path/child::n/descendant\text{-}or\text{-}self::m]\!]x
\end{aligned}$$

□

A.2 Descendant Axis

According to the Stripe projection for *descendant* axis, we have that for StripeSet $\mathcal{SS}_n$ the following holds:

$$\begin{aligned}
\mathsf{ss_nodes}(\mathcal{SS}_n) &= \{y \mid x \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge \mathsf{path}(x) \in \mathsf{proper_prefix}(\mathsf{path}(y)) \wedge \tau(y,n)\}\\
&= \{y \mid x \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge y \in \mathsf{children}^+(x) \wedge \tau(y,n)\} \qquad \text{(A.16)}
\end{aligned}$$

The following properties are deduced directly from the relationship among nodes:

$$x_2 \in \mathsf{children}^+(x_1) \wedge x_3 \in \mathsf{self}(x_2) \Rightarrow x_3 \in \mathsf{children}^+(x_1) \qquad \text{(A.17)}$$

$$x_2 \in \mathsf{children}^+(x_1) \wedge x_3 \in \mathsf{children}(x_2) \Rightarrow x_3 \in \mathsf{children}^+(x_1) \qquad \text{(A.18)}$$

$$x_2 \in \mathsf{children}^+(x_1) \wedge x_3 \in \mathsf{children}^+(x_2) \Rightarrow x_3 \in \mathsf{children}^+(x_1) \qquad \text{(A.19)}$$

$$x_2 \in \mathsf{children}^+(x_1) \wedge x_3 \in \mathsf{children}^*(x_2) \Rightarrow x_3 \in \mathsf{children}^+(x_1) \qquad \text{(A.20)}$$

Equivalence 2.13: For any path expression *path*, the following holds:

$$path/descendant::n/self::m \overset{\text{srx}}{\equiv} path/descendant::m$$

Proof: According to the Stripe projection for *self* axis, we have that for StripeSet $\mathcal{SS}_m$ the following holds:

$$\mathsf{ss\_nodes}(\mathcal{SS}_m) = \{\, x \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge \tau(x,m) \,\} \tag{A.21}$$

From Equations (A.16) , (A.21) we have that:

$$\forall z \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \exists x,y : x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \\ \wedge\, y \in \mathsf{children}^+(x) \wedge \tau(y,n) \wedge z = y \wedge \tau(z,m) \tag{A.22}$$

Now, for any context node x, we have:

$$\begin{aligned}
\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/descendant::m]\!](x,\mathcal{RS}_q) \\
&= \mathcal{V}(\mathcal{S}[\![path/descendant::m]\!]x,\mathcal{RS}_q) \\
&= \{\, x_3 \,|\, x_3 \in \mathcal{S}[\![path/descendant::m]\!]x \wedge x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q) \,\} \\
&= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![descendant::m]\!]x_1 \wedge \\
&\quad\; x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q) \,\} \\
&= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{children}^+(x_1) \wedge \tau(x_3,m) \wedge \\
&\quad\; x_3 \in \mathsf{ss_nodes}(\mathcal{SS}_m) \,\} \\
&\overset{(A.22)}{=} \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{children}^+(x_1) \wedge \tau(x_3,m) \wedge \\
&\quad\; x_1 \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}^+(x_1) \wedge \tau(x_2,n) \wedge \\
&\quad\; x_3 = x_2 \wedge \tau(x_3,m) \,\} \\
&\overset{(A.17)}{=} \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{children}^+(x_1) \wedge \tau(x_2,n) \wedge \\
&\quad\; x_3 = x_2 \wedge \tau(x_3,m) \,\} \\
&= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![descendant::n]\!]x_1 \wedge \\
&\quad\; x_3 \in \mathcal{S}[\![self::m]\!]x_2 \,\} \\
&= \mathcal{S}[\![path/descendant::n/self::m]\!]x
\end{aligned}$$

□

Equivalence 2.14: For any path expression *path*, the following holds:

$$path/descendant::n/child::m \overset{\text{srx}}{\equiv} path/descendant::m$$

Proof: According to the Stripe projection for *child* axis, we have that for StripeSet $\mathcal{SS}_m$ the following holds:

$$\begin{aligned}\mathsf{ss\_nodes}(\mathcal{SS}_m) &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge \mathsf{path}(x) = \mathsf{maximal\_prefix}(\mathsf{path}(y)) \wedge \tau(y,m) \,\} \\ &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge y \in \mathsf{children}(x) \wedge \tau(y,m) \,\} \qquad \text{(A.23)}\end{aligned}$$

From Equations (A.16) , (A.23) we have that:

$$\begin{aligned}\forall z \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \exists x, y : x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \\ \wedge\, y \in \mathsf{children}^+(x) \wedge \tau(y,n) \wedge z \in \mathsf{children}(y) \wedge \tau(z,m) \qquad \text{(A.24)}\end{aligned}$$

Now, for any context node x, we have:

$$\begin{aligned}\mathcal{F}[\![q']\!](x, \mathcal{RS}_q) &= \mathcal{F}[\![path/descendant{::}m]\!](x, \mathcal{RS}_q) \\ &= \mathcal{V}(\mathcal{S}[\![path/descendant{::}m]\!]x, \mathcal{RS}_q) \\ &= \{\, x_3 \,|\, x_3 \in \mathcal{S}[\![path/descendant{::}m]\!]x \wedge x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q) \,\} \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![descendant{::}m]\!]x_1 \wedge \\ &\quad\; x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q) \,\} \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{children}^+(x_1) \wedge \tau(x_3,m) \wedge \\ &\quad\; x_3 \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \,\} \\ &\overset{\text{(A.24)}}{=} \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{children}^+(x_1) \wedge \tau(x_3,m) \wedge \\ &\quad\; x_1 \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}^+(x_1) \wedge \tau(x_2,n) \wedge \\ &\quad\; x_3 \in \mathsf{children}(x_2) \wedge \tau(x_3,m) \,\} \\ &\overset{\text{(A.18)}}{=} \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{children}^+(x_1) \wedge \tau(x_2,n) \wedge \\ &\quad\; x_3 \in \mathsf{children}(x_2) \wedge \tau(x_3,m) \,\} \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![descendant{::}n]\!]x_1 \wedge \\ &\quad\; x_3 \in \mathcal{S}[\![child{::}m]\!]x_2 \,\} \\ &= \mathcal{S}[\![path/descendant{::}n/child{::}m]\!]x\end{aligned}$$

□

Equivalence 2.15: For any path expression *path*, the following holds:

$$path/descendant{::}n/descendant{::}m \overset{\mathsf{srx}}{\equiv} path/descendant{::}m$$

Proof: According to the Stripe projection for *descendant* axis, we have that for StripeSet

$\mathcal{SS}_m$ the following holds:

$$\begin{aligned}\mathsf{ss\_nodes}(\mathcal{SS}_m) &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge \mathsf{path}(x) \in \mathsf{proper\_prefix}(\mathsf{path}(y)) \wedge \tau(y,m) \,\} \\ &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge y \in \mathsf{children}^+(x) \wedge \tau(y,m) \,\} \qquad \text{(A.25)}\end{aligned}$$

From Equations (A.16) , (A.25) we have that:

$$\begin{aligned}\forall z \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \exists x,y : x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \\ \wedge\, y \in \mathsf{children}^+(x) \wedge \tau(y,n) \wedge z \in \mathsf{children}^+(y) \wedge \tau(z,m) \qquad \text{(A.26)}\end{aligned}$$

Now, for any context node x, we have:

$$\begin{aligned}\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/descendant{::}m]\!](x,\mathcal{RS}_q) \\ &= \mathcal{V}(\mathcal{S}[\![path/descendant{::}m]\!]x,\mathcal{RS}_q) \\ &= \{\, x_3 \,|\, x_3 \in \mathcal{S}[\![path/descendant{::}m]\!]x \wedge x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q) \,\} \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![descendant{::}m]\!]x_1 \wedge \\ &\qquad x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q) \,\} \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{children}^+(x_1) \wedge \tau(x_3,m) \wedge \\ &\qquad x_3 \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \,\} \\ &\overset{\text{(A.26)}}{=} \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{children}^+(x_1) \wedge \tau(x_3,m) \wedge \\ &\qquad x_1 \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}^+(x_1) \wedge \tau(x_2,n) \wedge \\ &\qquad x_3 \in \mathsf{children}^+(x_2) \wedge \tau(x_3,m) \,\} \\ &\overset{\text{(A.19)}}{=} \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{children}^+(x_1) \wedge \tau(x_2,n) \wedge \\ &\qquad x_3 \in \mathsf{children}^+(x_2) \wedge \tau(x_3,m) \,\} \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![descendant{::}n]\!]x_1 \wedge \\ &\qquad x_3 \in \mathcal{S}[\![descendant{::}m]\!]x_2 \,\} \\ &= \mathcal{S}[\![path/descendant{::}n/descendant{::}m]\!]x\end{aligned}$$

□

Equivalence 2.16: For any path expression *path*, the following holds:

$$path/descendant{::}n/descendant\text{-}or\text{-}self{::}m \overset{\mathsf{srx}}{\equiv} path/descendant{::}m$$

Proof: According to the Stripe projection for *descendant-or-self* axis, we have that for StripeSet $\mathcal{SS}_m$ the following holds:

$$\begin{aligned}\mathsf{ss\_nodes}(\mathcal{SS}_m) &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge \mathsf{path}(x) \in \mathsf{prefix}(\mathsf{path}(y)) \wedge \tau(y,m) \,\} \\ &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge y \in \mathsf{children}^*(x) \wedge \tau(y,m) \,\} \qquad \text{(A.27)}\end{aligned}$$

From Equations (A.16) , (A.27) we have that:

$$\forall z \in \mathsf{ss\_nodes}(\mathcal{SS}_m)\exists x,y : x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \\ \wedge y \in \mathsf{children}^+(x) \wedge \tau(y,n) \wedge z \in \mathsf{children}^*(y) \wedge \tau(z,m) \quad \text{(A.28)}$$

Now, for any context node x, we have:

$$\begin{aligned}
\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/descendant::m]\!](x,\mathcal{RS}_q) \\
&= \mathcal{V}(\mathcal{S}[\![path/descendant::m]\!]x,\mathcal{RS}_q) \\
&= \{x_3 \,|\, x_3 \in \mathcal{S}[\![path/descendant::m]\!]x \wedge x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q)\} \\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![descendant::m]\!]x_1 \wedge \\
&\quad x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q)\} \\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{children}^+(x_1) \wedge \tau(x_3,m) \wedge \\
&\quad x_3 \in \mathsf{ss_nodes}(\mathcal{SS}_m)\} \\
&\overset{\text{(A.28)}}{=} \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{children}^+(x_1) \wedge \tau(x_3,m) \wedge \\
&\quad x_1 \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}^+(x_1) \wedge \tau(x_2,n) \wedge \\
&\quad x_3 \in \mathsf{children}^*(x_2) \wedge \tau(x_3,m)\} \\
&\overset{\text{(A.20)}}{=} \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{children}^+(x_1) \wedge \tau(x_2,n) \wedge \\
&\quad x_3 \in \mathsf{children}^*(x_2) \wedge \tau(x_3,m)\} \\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![descendant::n]\!]x_1 \wedge \\
&\quad x_3 \in \mathcal{S}[\![descendant\text{-}or\text{-}self::m]\!]x_2\} \\
&= \mathcal{S}[\![path/descendant::n/descendant\text{-}or\text{-}self::m]\!]x
\end{aligned}$$

□

A.3 Descendant-or-self Axis

According to the Stripe projection for *descendant-or-self* axis, we have that for StripeSet $\mathcal{SS}_n$ the following holds:

$$\begin{aligned}
\mathsf{ss_nodes}(\mathcal{SS}_n) &= \{y \,|\, x \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge \mathsf{path}(x) \in \mathsf{prefix}(\mathsf{path}(y)) \wedge \tau(y,n)\} \\
&= \{y \,|\, x \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge y \in \mathsf{children}^*(x) \wedge \tau(y,n)\} \qquad \text{(A.29)}
\end{aligned}$$

The following properties are deduced directly from the relationship among nodes:

$$x_2 \in \mathsf{children}^*(x_1) \wedge x_3 \in \mathsf{self}(x_2) \Rightarrow x_3 \in \mathsf{children}^*(x_1) \tag{A.30}$$

$$x_2 \in \mathsf{children}^*(x_1) \wedge x_3 \in \mathsf{children}(x_2) \Rightarrow x_3 \in \mathsf{children}^+(x_1) \tag{A.31}$$

$$x_2 \in \mathsf{children}^*(x_1) \wedge x_3 \in \mathsf{children}^+(x_2) \Rightarrow x_3 \in \mathsf{children}^+(x_1) \tag{A.32}$$

$$x_2 \in \mathsf{children}^*(x_1) \wedge x_3 \in \mathsf{children}^*(x_2) \Rightarrow x_3 \in \mathsf{children}^*(x_1) \tag{A.33}$$

Equivalence 2.17: For any path expression *path*, the following holds:

$$path/descendant\text{-}or\text{-}self::n/self::m \overset{\mathsf{srx}}{\equiv} path/descendant\text{-}or\text{-}self::m$$

Proof: According to the Stripe projection for *self* axis, we have that for StripeSet $\mathcal{SS}_m$ the following holds:

$$\mathsf{ss\_nodes}(\mathcal{SS}_m) = \{\, x \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge \tau(x,m) \,\} \tag{A.34}$$

From Equations (A.29) , (A.34) we have that:

$$\forall z \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \exists x,y : x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge y \in \mathsf{children}^*(x) \wedge \tau(y,n) \wedge z = y \wedge \tau(z,m) \tag{A.35}$$

Now, for any context node x, we have:

$$\begin{aligned}
\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/descendant\text{-}or\text{-}self::m]\!](x,\mathcal{RS}_q)\\
&= \mathcal{V}(\mathcal{S}[\![path/descendant\text{-}or\text{-}self::m]\!]x,\mathcal{RS}_q)\\
&= \{x_3 \mid x_3 \in \mathcal{S}[\![path/descendant\text{-}or\text{-}self::m]\!]x \wedge\\
&\quad x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![descendant\text{-}or\text{-}self::m]\!]x_1 \wedge\\
&\quad x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{children}^*(x_1) \wedge \tau(x_3,m) \wedge\\
&\quad x_3 \in \mathsf{ss_nodes}(\mathcal{SS}_m)\}\\
&\overset{(A.35)}{=} \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{children}^+(x_1) \wedge \tau(x_3,m) \wedge\\
&\quad x_1 \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}^*(x_1) \wedge \tau(x_2,n) \wedge\\
&\quad x_3 = x_2 \wedge \tau(x_3,m)\}\\
&\overset{(A.30)}{=} \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{children}^*(x_1) \wedge \tau(x_2,n) \wedge\\
&\quad x_3 = x_2 \wedge \tau(x_3,m)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![descendant\text{-}or\text{-}self::n]\!]x_1 \wedge\\
&\quad x_3 \in \mathcal{S}[\![self::m]\!]x_2\}\\
&= \mathcal{S}[\![path/descendant\text{-}or\text{-}self::n/self::m]\!]x
\end{aligned}$$

□

Equivalence 2.18: For any path expression *path*, the following holds:

$$path/descendant\text{-}or\text{-}self::n/child::m \overset{\mathsf{srx}}{\equiv} path/descendant::m$$

Proof: According to the Stripe projection for *child* axis, we have that for StripeSet $\mathcal{SS}_m$ the following holds:

$$\begin{aligned}
\mathsf{ss_nodes}(\mathcal{SS}_m) &= \{y \mid x \in \mathsf{ss_nodes}(\mathcal{SS}_n) \wedge \mathsf{path}(x) = \mathsf{maximal_prefix}(\mathsf{path}(y)) \wedge \tau(y,m)\}\\
&= \{y \mid x \in \mathsf{ss_nodes}(\mathcal{SS}_n) \wedge y \in \mathsf{children}(x) \wedge \tau(y,m)\} \qquad (A.36)
\end{aligned}$$

From Equations (A.29) , (A.36) we have that:

$$\begin{aligned}
\forall z \in \mathsf{ss_nodes}(\mathcal{SS}_m) \exists x,y : x \in \mathsf{ss_nodes}(\mathcal{SS}_{path})\\
\wedge y \in \mathsf{children}^*(x) \wedge \tau(y,n) \wedge z \in \mathsf{children}(y) \wedge \tau(z,m) \qquad (A.37)
\end{aligned}$$

Now, for any context node x, we have:

$$\begin{aligned}
\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/descendant{::}m]\!](x,\mathcal{RS}_q)\\
&= \mathcal{V}(\mathcal{S}[\![path/descendant{::}m]\!]x,\mathcal{RS}_q)\\
&= \{x_3 \,|\, x_3 \in \mathcal{S}[\![path/descendant{::}m]\!]x \wedge x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![descendant{::}m]\!]x_1 \wedge\\
&\quad\; x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{children}^+(x_1) \wedge \tau(x_3,m) \wedge\\
&\quad\; x_3 \in \mathsf{ss_nodes}(\mathcal{SS}_m)\}\\
&\overset{(A.37)}{=} \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{children}^+(x_1) \wedge \tau(x_3,m) \wedge\\
&\quad\; x_1 \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}^*(x_1) \wedge \tau(x_2,n) \wedge\\
&\quad\; x_3 \in \mathsf{children}(x_2) \wedge \tau(x_3,m)\}\\
&\overset{(A.31)}{=} \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{children}^*(x_1) \wedge \tau(x_2,n) \wedge\\
&\quad\; x_3 \in \mathsf{children}(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![descendant\text{-}or\text{-}self{::}n]\!]x_1 \wedge\\
&\quad\; x_3 \in \mathcal{S}[\![child{::}m]\!]x_2\}\\
&= \mathcal{S}[\![path/descendant\text{-}or\text{-}self{::}n/child{::}m]\!]x
\end{aligned}$$

□

Equivalence 2.19: For any path expression *path*, the following holds:

$$path/descendant\text{-}or\text{-}self{::}n/descendant{::}m \overset{\mathsf{srx}}{\equiv} path/descendant{::}m$$

Proof: According to the Stripe projection for *descendant* axis, we have that for StripeSet $\mathcal{SS}_m$ the following holds:

$$\begin{aligned}
\mathsf{ss_nodes}(\mathcal{SS}_m) &= \{y \,|\, x \in \mathsf{ss_nodes}(\mathcal{SS}_n) \wedge \mathsf{path}(x) \in \mathsf{proper_prefix}(\mathsf{path}(y)) \wedge \tau(y,m)\}\\
&= \{y \,|\, x \in \mathsf{ss_nodes}(\mathcal{SS}_n) \wedge y \in \mathsf{children}^+(x) \wedge \tau(y,m)\} \qquad (A.38)
\end{aligned}$$

From Equations (A.29) , (A.38) we have that:

$$\begin{aligned}
\forall z \in \mathsf{ss_nodes}(\mathcal{SS}_m) \exists x,y : x \in \mathsf{ss_nodes}(\mathcal{SS}_{path})\\
\wedge y \in \mathsf{children}^*(x) \wedge \tau(y,n) \wedge z \in \mathsf{children}^+(y) \wedge \tau(z,m) \quad (A.39)
\end{aligned}$$

Now, for any context node x, we have:

$$\begin{aligned}
\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/descendant{::}m]\!](x,\mathcal{RS}_q)\\
&= \mathcal{V}(\mathcal{S}[\![path/descendant{::}m]\!]x,\mathcal{RS}_q)\\
&= \{x_3 \,|\, x_3 \in \mathcal{S}[\![path/descendant{::}m]\!]x \wedge x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![descendant{::}m]\!]x_1 \wedge\\
&\quad\; x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{children}^+(x_1) \wedge \tau(x_3,m) \wedge\\
&\quad\; x_3 \in \mathsf{ss_nodes}(\mathcal{SS}_m)\}\\
&\overset{(A.39)}{=} \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{children}^+(x_1) \wedge \tau(x_3,m) \wedge\\
&\quad\; x_1 \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}^*(x_1) \wedge \tau(x_2,n) \wedge\\
&\quad\; x_3 \in \mathsf{children}^+(x_2) \wedge \tau(x_3,m)\}\\
&\overset{(A.32)}{=} \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{children}^*(x_1) \wedge \tau(x_2,n) \wedge\\
&\quad\; x_3 \in \mathsf{children}^+(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![descendant\text{-}or\text{-}self{::}n]\!]x_1 \wedge\\
&\quad\; x_3 \in \mathcal{S}[\![descendant{::}m]\!]x_2\}\\
&= \mathcal{S}[\![path/descendant\text{-}or\text{-}self{::}n/descendant{::}m]\!]x
\end{aligned}$$

□

Equivalence 2.20: For any path expression $path$, the following holds:

$$path/descendant\text{-}or\text{-}self{::}n/descendant\text{-}or\text{-}self{::}m \overset{\mathsf{srx}}{\equiv} path/descendant\text{-}or\text{-}self{::}m$$

Proof: According to the Stripe projection for *descendant-or-self* axis, we have that for StripeSet $\mathcal{SS}_m$ the following holds:

$$\begin{aligned}
\mathsf{ss_nodes}(\mathcal{SS}_m) &= \{y \,|\, x \in \mathsf{ss_nodes}(\mathcal{SS}_n) \wedge \mathsf{path}(x) \in \mathsf{prefix}(\mathsf{path}(y)) \wedge \tau(y,m)\}\\
&= \{y \,|\, x \in \mathsf{ss_nodes}(\mathcal{SS}_n) \wedge y \in \mathsf{children}^*(x) \wedge \tau(y,m)\} \qquad \text{(A.40)}
\end{aligned}$$

From Equations (A.29) , (A.40) we have that:

$$\begin{aligned}
\forall z \in \mathsf{ss_nodes}(\mathcal{SS}_m) \exists x,y : x \in \mathsf{ss_nodes}(\mathcal{SS}_{path})\\
\wedge y \in \mathsf{children}^*(x) \wedge \tau(y,n) \wedge z \in \mathsf{children}^*(y) \wedge \tau(z,m) \quad \text{(A.41)}
\end{aligned}$$

Now, for any context node x, we have:

$$\begin{aligned}
\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/descendant\text{-}or\text{-}self::m]\!](x,\mathcal{RS}_q)\\
&= \mathcal{V}(\mathcal{S}[\![path/descendant\text{-}or\text{-}self::m]\!]x,\mathcal{RS}_q)\\
&= \{x_3 \,|\, x_3 \in \mathcal{S}[\![path/descendant\text{-}or\text{-}self::m]\!]x \wedge x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![descendant\text{-}or\text{-}self::m]\!]x_1 \wedge\\
&\quad\ x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{children}^*(x_1) \wedge \tau(x_3,m) \wedge\\
&\quad\ x_3 \in \mathsf{ss_nodes}(\mathcal{SS}_m)\}\\
&\overset{(A.41)}{=} \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{children}^*(x_1) \wedge \tau(x_3,m) \wedge\\
&\quad\ x_1 \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}^*(x_1) \wedge \tau(x_2,n) \wedge\\
&\quad\ x_3 \in \mathsf{children}^*(x_2) \wedge \tau(x_3,m)\}\\
&\overset{(A.33)}{=} \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{children}^*(x_1) \wedge \tau(x_2,n) \wedge\\
&\quad\ x_3 \in \mathsf{children}^*(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![descendant\text{-}or\text{-}self::n]\!]x_1 \wedge\\
&\quad\ x_3 \in \mathcal{S}[\![descendant\text{-}or\text{-}self::m]\!]x_2\}\\
&= \mathcal{S}[\![path/descendant\text{-}or\text{-}self::n/descendant\text{-}or\text{-}self::m]\!]x
\end{aligned}$$

□

A.4 Parent Axis

According to the Stripe projection for *parent* axis, we have that for StripeSet $\mathcal{SS}_n$ the following holds:

$$\begin{aligned}
\mathsf{ss_nodes}(\mathcal{SS}_n) &= \{y \,|\, x \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge\\
&\qquad \mathsf{path}(y) = \mathsf{maximal_prefix}(\mathsf{path}(x)) \wedge \tau(y,n)\} \qquad (A.42)\\
&= \{y \,|\, x \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge y \in \mathsf{parent}(x) \wedge \tau(y,n)\}
\end{aligned}$$

The following properties are deduced directly from the relationship among nodes:

$$x_2 \in \mathsf{parent}(x_1) \wedge x_3 \in \mathsf{self}(x_2) \Rightarrow x_3 \in \mathsf{parent}(x_1) \qquad (A.43)$$

$$x_2 \in \mathsf{parent}(x_1) \wedge x_3 \in \mathsf{parent}(x_2) \Rightarrow x_3 \in \mathsf{parent}^+(x_1) \qquad (A.44)$$

$$x_2 \in \mathsf{parent}(x_1) \wedge x_3 \in \mathsf{parent}^+(x_2) \Rightarrow x_3 \in \mathsf{parent}^+(x_1) \qquad (A.45)$$

$$x_2 \in \mathsf{parent}(x_1) \wedge x_3 \in \mathsf{parent}^*(x_2) \Rightarrow x_3 \in \mathsf{parent}^+(x_1) \qquad (A.46)$$

Equivalence 2.21: For any path expression *path*, the following holds:

$$path/parent::n/self::m \overset{\text{srx}}{\equiv} path/parent::m$$

Proof: According to the Stripe projection for *self* axis, we have that for StripeSet $\mathcal{SS}_m$ the following holds:

$$\mathsf{ss\_nodes}(\mathcal{SS}_m) = \{\, x \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge \tau(x,m) \,\} \tag{A.47}$$

From Equations (A.42) , (A.47) we have that:

$$\forall z \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \exists x,y : x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge y \in \mathsf{parent}(x) \wedge \tau(y,n) \wedge z = y \wedge \tau(z,m) \tag{A.48}$$

Now, for any context node x, we have:

$$\begin{aligned}
\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/parent::m]\!](x,\mathcal{RS}_q) \\
&= \mathcal{V}(\mathcal{S}[\![path/parent::m]\!]x,\mathcal{RS}_q) \\
&= \{\, x_3 \,|\, x_3 \in \mathcal{S}[\![path/parent::m]\!]x \wedge x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q) \,\} \\
&= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![parent::m]\!]x_1 \wedge \\
&\quad\; x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q) \,\} \\
&= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{parent}(x_1) \wedge \tau(x_3,m) \wedge \\
&\quad\; x_3 \in \mathsf{ss_nodes}(\mathcal{SS}_m) \,\} \\
&\overset{(A.48)}{=} \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{parent}(x_1) \wedge \tau(x_3,m) \wedge \\
&\quad\; x_1 \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{parent}(x_1) \wedge \tau(x_2,n) \wedge \\
&\quad\; x_3 = x_2 \wedge \tau(x_3,m) \,\} \\
&\overset{(A.43)}{=} \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{parent}(x_1) \wedge \tau(x_2,n) \wedge \\
&\quad\; x_3 = x_2 \wedge \tau(x_3,m) \,\} \\
&= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![parent::n]\!]x_1 \wedge x_3 \in \mathcal{S}[\![self::m]\!]x_2 \,\} \\
&= \mathcal{S}[\![path/parent::n/self::m]\!]x
\end{aligned}$$

□

Equivalence 2.22: For any path expression *path*, the following holds:

$$path/parent::n/parent::m \overset{\text{srx}}{\equiv} path/ancestor::m$$

Proof: According to the Stripe projection for *parent* axis, we have that for StripeSet $\mathcal{SS}_m$ the following holds:

$$\begin{aligned}\mathsf{ss\_nodes}(\mathcal{SS}_m) &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge \mathsf{path}(y) = \mathsf{maximal\_prefix}(\mathsf{path}(x)) \wedge \tau(y,m) \,\} \\ &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge y \in \mathsf{parent}(x) \wedge \tau(y,m) \,\} \qquad \text{(A.49)}\end{aligned}$$

From Equations (A.42) , (A.49) we have that:

$$\begin{aligned}\forall z \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \exists x, y : x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \\ \wedge\, y \in \mathsf{parent}(x) \wedge \tau(y,n) \wedge z \in \mathsf{parent}(y) \wedge \tau(z,m) \qquad \text{(A.50)}\end{aligned}$$

Now, for any context node x, we have:

$$\begin{aligned}\mathcal{F}[\![q']\!](x, \mathcal{RS}_q) &= \mathcal{F}[\![path/ancestor{::}m]\!](x, \mathcal{RS}_q) \\ &= \mathcal{V}(\mathcal{S}[\![path/ancestor{::}m]\!]x, \mathcal{RS}_q) \\ &= \{\, x_3 \,|\, x_3 \in \mathcal{S}[\![path/ancestor{::}m]\!]x \wedge x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q) \,\} \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![ancestor{::}m]\!]x_1 \wedge \\ &\qquad x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q) \,\} \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{parent}^+(x_1) \wedge \tau(x_3,m) \wedge \\ &\qquad x_3 \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \,\} \\ &\overset{\text{(A.50)}}{=} \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{parent}^+(x_1) \wedge \tau(x_3,m) \wedge \\ &\qquad x_1 \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{parent}(x_1) \wedge \tau(x_2,n) \wedge \\ &\qquad x_3 \in \mathsf{parent}(x_2) \wedge \tau(x_3,m) \,\} \\ &\overset{\text{(A.44)}}{=} \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{parent}(x_1) \wedge \tau(x_2,n) \wedge \\ &\qquad x_3 \in \mathsf{parent}(x_2) \wedge \tau(x_3,m) \,\} \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![parent{::}n]\!]x_1 \wedge x_3 \in \mathcal{S}[\![parent{::}m]\!]x_2 \,\} \\ &= \mathcal{S}[\![path/parent{::}n/parent{::}m]\!]x\end{aligned}$$

□

Equivalence 2.23: For any path expression *path*, the following holds:

$$path/parent{::}n/ancestor{::}m \overset{\mathsf{srx}}{\equiv} path/ancestor{::}m$$

Proof: According to the Stripe projection for *ancestor* axis, we have that for StripeSet $\mathcal{SS}_m$ the following holds:

$$\begin{aligned}\mathsf{ss\_nodes}(\mathcal{SS}_m) &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge \mathsf{path}(y) \in \mathsf{proper\_prefix}(\mathsf{path}(x)) \wedge \tau(y,m) \,\} \\ &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge y \in \mathsf{parent}^+(x) \wedge \tau(y,m) \,\} \qquad \text{(A.51)}\end{aligned}$$

From Equations (A.42) , (A.51) we have that:

$$\forall z \in \mathsf{ss\_nodes}(\mathcal{SS}_m)\exists x,y : x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \\ \wedge y \in \mathsf{parent}(x) \wedge \tau(y,n) \wedge z \in \mathsf{parent}^+(y) \wedge \tau(z,m) \quad \text{(A.52)}$$

Now, for any context node x, we have:

$$\begin{aligned}
\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/ancestor::m]\!](x,\mathcal{RS}_q) \\
&= \mathcal{V}(\mathcal{S}[\![path/ancestor::m]\!]x,\mathcal{RS}_q) \\
&= \{x_3 \,|\, x_3 \in \mathcal{S}[\![path/ancestor::m]\!]x \wedge x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q)\} \\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![ancestor::m]\!]x_1 \wedge \\
&\quad x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q)\} \\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{parent}^+(x_1) \wedge \tau(x_3,m) \wedge \\
&\quad x_3 \in \mathsf{ss_nodes}(\mathcal{SS}_m)\} \\
&\overset{\text{(A.52)}}{=} \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{parent}^+(x_1) \wedge \tau(x_3,m) \wedge \\
&\quad x_1 \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{parent}(x_1) \wedge \tau(x_2,n) \wedge \\
&\quad x_3 \in \mathsf{parent}^+(x_2) \wedge \tau(x_3,m)\} \\
&\overset{\text{(A.45)}}{=} \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{parent}(x_1) \wedge \tau(x_2,n) \wedge \\
&\quad x_3 \in \mathsf{parent}^+(x_2) \wedge \tau(x_3,m)\} \\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![parent::n]\!]x_1 \wedge \\
&\quad x_3 \in \mathcal{S}[\![ancestor::m]\!]x_2\} \\
&= \mathcal{S}[\![path/parent::n/ancestor::m]\!]x
\end{aligned}$$

□

Equivalence 2.24: For any path expression *path*, the following holds:

$$path/parent::n/ancestor\text{-}or\text{-self}::m \overset{\mathsf{srx}}{\equiv} path/ancestor::m$$

Proof: According to the Stripe projection for *ancestor-or*-self axis, we have that for StripeSet $\mathcal{SS}_m$ the following holds:

$$\begin{aligned}
\mathsf{ss_nodes}(\mathcal{SS}_m) &= \{y \,|\, x \in \mathsf{ss_nodes}(\mathcal{SS}_n) \wedge \mathsf{path}(y) \in \mathsf{prefix}(\mathsf{path}(x)) \wedge \tau(y,m)\} \\
&= \{y \,|\, x \in \mathsf{ss_nodes}(\mathcal{SS}_n) \wedge y \in \mathsf{parent}^*(x) \wedge \tau(y,m)\} \quad \text{(A.53)}
\end{aligned}$$

From Equations (A.42) , (A.53) we have that:

$$\forall z \in \mathsf{ss\_nodes}(\mathcal{SS}_m)\exists x,y : x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \\ \wedge y \in \mathsf{parent}(x) \wedge \tau(y,n) \wedge z \in \mathsf{parent}^*(y) \wedge \tau(z,m) \quad \text{(A.54)}$$

Now, for any context node x, we have:

$$\begin{aligned}
\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/ancestor{::}m]\!](x,\mathcal{RS}_q)\\
&= \mathcal{V}(\mathcal{S}[\![path/ancestor{::}m]\!]x,\mathcal{RS}_q)\\
&= \{x_3\,|\,x_3 \in \mathcal{S}[\![path/ancestor{::}m]\!]x \wedge x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3\,|\,x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![ancestor{::}m]\!]x_1 \wedge\\
&\quad\ x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3\,|\,x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{parent}^+(x_1) \wedge \tau(x_3,m) \wedge\\
&\quad\ x_3 \in \mathsf{ss_nodes}(\mathcal{SS}_m)\}\\
&\overset{(A.54)}{=} \{x_3\,|\,x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{parent}^+(x_1) \wedge \tau(x_3,m) \wedge\\
&\quad\ x_1 \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{parent}(x_1) \wedge \tau(x_2,n) \wedge\\
&\quad\ x_3 \in \mathsf{parent}^*(x_2) \wedge \tau(x_3,m)\}\\
&\overset{(A.46)}{=} \{x_3\,|\,x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{parent}(x_1) \wedge \tau(x_2,n) \wedge\\
&\quad\ x_3 \in \mathsf{parent}^*(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3\,|\,x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![parent{::}n]\!]x_1 \wedge\\
&\quad\ x_3 \in \mathcal{S}[\![ancestor\text{-}or\text{-}\mathrm{self}{::}m]\!]x_2\}\\
&= \mathcal{S}[\![path/parent{::}n/ancestor\text{-}or\text{-}\mathrm{self}{::}m]\!]x
\end{aligned}$$

□

A.5 Ancestor Axis

According to the Stripe projection for *ancestor* axis, we have that for StripeSet $\mathcal{SS}_n$ the following holds:

$$\begin{aligned}
\mathsf{ss_nodes}(\mathcal{SS}_n) &= \{y\,|\,x \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge \mathsf{path}(y) \in \mathsf{proper_prefix}(\mathsf{path}(x)) \wedge \tau(y,n)\}\\
&= \{y\,|\,x \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge y \in \mathsf{parent}^+(x) \wedge \tau(y,n)\} \qquad \text{(A.55)}
\end{aligned}$$

The following properties are deduced directly from the relationship among nodes:

$$x_2 \in \mathsf{parent}^+(x_1) \wedge x_3 \in \mathsf{self}(x_2) \Rightarrow x_3 \in \mathsf{parent}^+(x_1) \qquad \text{(A.56)}$$

$$x_2 \in \mathsf{parent}^+(x_1) \wedge x_3 \in \mathsf{parent}(x_2) \Rightarrow x_3 \in \mathsf{parent}^+(x_1) \qquad \text{(A.57)}$$

$$x_2 \in \mathsf{parent}^+(x_1) \wedge x_3 \in \mathsf{parent}^+(x_2) \Rightarrow x_3 \in \mathsf{parent}^+(x_1) \qquad \text{(A.58)}$$

$$x_2 \in \mathsf{parent}^+(x_1) \wedge x_3 \in \mathsf{parent}^*(x_2) \Rightarrow x_3 \in \mathsf{parent}^+(x_1) \qquad \text{(A.59)}$$

Equivalence 2.25: For any path expression *path*, the following holds:

$$path/ancestor::n/self::m \overset{\text{srx}}{\equiv} path/ancestor::m$$

Proof: According to the Stripe projection for *self* axis, we have that for StripeSet $\mathcal{SS}_m$ the following holds:

$$\mathsf{ss\_nodes}(\mathcal{SS}_m) = \{\, x \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge \tau(x,m) \,\} \tag{A.60}$$

From Equations (A.55) , (A.60) we have that:

$$\forall z \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \exists x,y : x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge y \in \mathsf{parent}^+(x) \wedge \tau(y,n) \wedge z = y \wedge \tau(z,m) \tag{A.61}$$

Now, for any context node x, we have:

$$\begin{aligned}
\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/ancestor::m]\!](x,\mathcal{RS}_q) \\
&= \mathcal{V}(\mathcal{S}[\![path/ancestor::m]\!]x,\mathcal{RS}_q) \\
&= \{\, x_3 \,|\, x_3 \in \mathcal{S}[\![path/ancestor::m]\!]x \wedge x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q) \,\} \\
&= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![ancestor::m]\!]x_1 \wedge \\
&\quad\; x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q) \,\} \\
&= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{parent}^+(x_1) \wedge \tau(x_3,m) \wedge \\
&\quad\; x_3 \in \mathsf{ss_nodes}(\mathcal{SS}_m) \,\} \\
&\overset{(A.61)}{=} \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{parent}^+(x_1) \wedge \tau(x_3,m) \wedge \\
&\quad\; x_1 \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{parent}^+(x_1) \wedge \tau(x_2,n) \wedge \\
&\quad\; x_3 = x_2 \wedge \tau(x_3,m) \,\} \\
&\overset{(A.56)}{=} \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{parent}^+(x_1) \wedge \tau(x_2,n) \wedge \\
&\quad\; x_3 = x_2 \wedge \tau(x_3,m) \,\} \\
&= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![ancestor::n]\!]x_1 \wedge x_3 \in \mathcal{S}[\![self::m]\!]x_2 \,\} \\
&= \mathcal{S}[\![path/ancestor::n/self::m]\!]x
\end{aligned}$$

□

Equivalence 2.26: For any path expression *path*, the following holds:

$$path/ancestor::n/parent::m \overset{\text{srx}}{\equiv} path/ancestor::m$$

Proof: According to the Stripe projection for *parent* axis, we have that for StripeSet $\mathcal{SS}_m$ the following holds:

$$\begin{aligned}\mathsf{ss\_nodes}(\mathcal{SS}_m) &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge \mathsf{path}(y) = \mathsf{maximal\_prefix}(\mathsf{path}(x)) \wedge \tau(y,m) \,\} \\ &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge y \in \mathsf{parent}(x) \wedge \tau(y,m) \,\} \qquad \text{(A.62)}\end{aligned}$$

From Equations (A.55) , (A.62) we have that:

$$\begin{aligned}\forall z \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \exists x,y : x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \\ \wedge\, y \in \mathsf{parent}^+(x) \wedge \tau(y,n) \wedge z \in \mathsf{parent}(y) \wedge \tau(z,m) \qquad \text{(A.63)}\end{aligned}$$

Now, for any context node x, we have:

$$\begin{aligned}\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/ancestor::m]\!](x,\mathcal{RS}_q) \\ &= \mathcal{V}(\mathcal{S}[\![path/ancestor::m]\!]x,\mathcal{RS}_q) \\ &= \{\, x_3 \,|\, x_3 \in \mathcal{S}[\![path/ancestor::m]\!]x \wedge x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q) \,\} \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![ancestor::m]\!]x_1 \wedge \\ &\quad\; x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q) \,\} \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{parent}^+(x_1) \wedge \tau(x_3,m) \wedge \\ &\quad\; x_3 \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \,\} \\ &\overset{\text{(A.63)}}{=} \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{parent}^+(x_1) \wedge \tau(x_3,m) \wedge \\ &\quad\; x_1 \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{parent}^+(x_1) \wedge \tau(x_2,n) \wedge \\ &\quad\; x_3 \in \mathsf{parent}(x_2) \wedge \tau(x_3,m) \,\} \\ &\overset{\text{(A.57)}}{=} \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{parent}^+(x_1) \wedge \tau(x_2,n) \wedge \\ &\quad\; x_3 \in \mathsf{parent}(x_2) \wedge \tau(x_3,m) \,\} \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![ancestor::n]\!]x_1 \wedge \\ &\quad\; x_3 \in \mathcal{S}[\![parent::m]\!]x_2 \,\} \\ &= \mathcal{S}[\![path/ancestor::n/parent::m]\!]x\end{aligned}$$

□

Equivalence 2.27: For any path expression *path*, the following holds:

$$path/ancestor::n/ancestor::m \overset{\mathsf{srx}}{\equiv} path/ancestor::m$$

Proof: According to the Stripe projection for *ancestor* axis, we have that for StripeSet

$\mathcal{SS}_m$ the following holds:

$$\begin{aligned}\mathsf{ss\_nodes}(\mathcal{SS}_m) &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge \mathsf{path}(y) \in \mathsf{proper\_prefix}(\mathsf{path}(x)) \wedge \tau(y,m) \,\} \\ &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge y \in \mathsf{parent}^+(x) \wedge \tau(y,m) \,\} \qquad \text{(A.64)}\end{aligned}$$

From Equations (A.55) , (A.64) we have that:

$$\begin{aligned}\forall z \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \exists x,y : x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \\ \wedge\, y \in \mathsf{parent}^+(x) \wedge \tau(y,n) \wedge z \in \mathsf{parent}^+(y) \wedge \tau(z,m) \qquad \text{(A.65)}\end{aligned}$$

Now, for any context node x, we have:

$$\begin{aligned}\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/ancestor::m]\!](x,\mathcal{RS}_q) \\ &= \mathcal{V}(\mathcal{S}[\![path/ancestor::m]\!]x,\mathcal{RS}_q) \\ &= \{\, x_3 \,|\, x_3 \in \mathcal{S}[\![path/ancestor::m]\!]x \wedge x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q) \,\} \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![ancestor::m]\!]x_1 \wedge \\ &\qquad x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q) \,\} \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{parent}^+(x_1) \wedge \tau(x_3,m) \wedge \\ &\qquad x_3 \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \,\} \\ &\overset{\text{(A.65)}}{=} \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{parent}^+(x_1) \wedge \tau(x_3,m) \wedge \\ &\qquad x_1 \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{parent}^+(x_1) \wedge \tau(x_2,n) \wedge \\ &\qquad x_3 \in \mathsf{parent}^+(x_2) \wedge \tau(x_3,m) \,\} \\ &\overset{\text{(A.58)}}{=} \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{parent}^+(x_1) \wedge \tau(x_2,n) \wedge \\ &\qquad x_3 \in \mathsf{parent}^+(x_2) \wedge \tau(x_3,m) \,\} \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![ancestor::n]\!]x_1 \wedge \\ &\qquad x_3 \in \mathcal{S}[\![ancestor::m]\!]x_2 \,\} \\ &= \mathcal{S}[\![path/ancestor::n/ancestor::m]\!]x\end{aligned}$$

□

Equivalence 2.28: For any path expression *path*, the following holds:

$$path/ancestor::n/ancestor\text{-}or\text{-self}::m \overset{\mathsf{srx}}{\equiv} path/ancestor::m$$

Proof: According to the Stripe projection for *ancestor-or*-self axis, we have that for StripeSet $\mathcal{SS}_m$ the following holds:

$$\begin{aligned}\mathsf{ss\_nodes}(\mathcal{SS}_m) &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge \mathsf{path}(y) \in \mathsf{prefix}(\mathsf{path}(x)) \wedge \tau(y,m) \,\} \\ &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge y \in \mathsf{parent}^*(x) \wedge \tau(y,m) \,\} \qquad \text{(A.66)}\end{aligned}$$

From Equations (A.55) , (A.66) we have that:

$$\forall z \in \mathsf{ss\_nodes}(\mathcal{SS}_m)\exists x,y : x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \\ \wedge y \in \mathsf{parent}^+(x) \wedge \tau(y,n) \wedge z \in \mathsf{parent}^*(y) \wedge \tau(z,m) \quad \text{(A.67)}$$

Now, for any context node x, we have:

$$\begin{aligned}
\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/ancestor::m]\!](x,\mathcal{RS}_q) \\
&= \mathcal{V}(\mathcal{S}[\![path/ancestor::m]\!]x,\mathcal{RS}_q) \\
&= \{x_3 \,|\, x_3 \in \mathcal{S}[\![path/ancestor::m]\!]x \wedge x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q)\} \\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![ancestor::m]\!]x_1 \wedge \\
&\quad\; x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q)\} \\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{parent}^+(x_1) \wedge \tau(x_3,m) \wedge \\
&\quad\; x_3 \in \mathsf{ss_nodes}(\mathcal{SS}_m)\} \\
&\overset{(A.67)}{=} \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{parent}^+(x_1) \wedge \tau(x_3,m) \wedge \\
&\quad\; x_1 \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{parent}^+(x_1) \wedge \tau(x_2,n) \wedge \\
&\quad\; x_3 \in \mathsf{parent}^*(x_2) \wedge \tau(x_3,m)\} \\
&\overset{(A.59)}{=} \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{parent}^+(x_1) \wedge \tau(x_2,n) \wedge \\
&\quad\; x_3 \in \mathsf{parent}^*(x_2) \wedge \tau(x_3,m)\} \\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![ancestor::n]\!]x_1 \wedge \\
&\quad\; x_3 \in \mathcal{S}[\![ancestor\text{-}or\text{-}self::m]\!]x_2\} \\
&= \mathcal{S}[\![path/ancestor::n/ancestor\text{-}or\text{-}self::m]\!]x
\end{aligned}$$

□

A.6 Ancestor-or-self Axis

According to the Stripe projection for *ancestor-or*-self axis, we have that for StripeSet $\mathcal{SS}_n$ the following holds:

$$\begin{aligned}
\mathsf{ss_nodes}(\mathcal{SS}_n) &= \{y \,|\, x \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge \mathsf{path}(y) \in \mathsf{prefix}(\mathsf{path}(x)) \wedge \tau(y,n)\} \\
&= \{y \,|\, x \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge y \in \mathsf{parent}^*(x) \wedge \tau(y,n)\} \qquad \text{(A.68)}
\end{aligned}$$

The following properties are deduced directly from the relationship among nodes:

$$x_2 \in \mathsf{parent}^*(x_1) \wedge x_3 \in \mathsf{self}(x_2) \Rightarrow x_3 \in \mathsf{parent}^*(x_1) \tag{A.69}$$

$$x_2 \in \mathsf{parent}^*(x_1) \wedge x_3 \in \mathsf{parent}(x_2) \Rightarrow x_3 \in \mathsf{parent}^+(x_1) \tag{A.70}$$

$$x_2 \in \mathsf{parent}^*(x_1) \wedge x_3 \in \mathsf{parent}^+(x_2) \Rightarrow x_3 \in \mathsf{parent}^+(x_1) \tag{A.71}$$

$$x_2 \in \mathsf{parent}^*(x_1) \wedge x_3 \in \mathsf{parent}^*(x_2) \Rightarrow x_3 \in \mathsf{parent}^*(x_1) \tag{A.72}$$

Equivalence 2.29: For any path expression *path*, the following holds:

$$path/ancestor\text{-}or\text{-self}::n/self::m \overset{\mathsf{srx}}{\equiv} path/ancestor\text{-}or\text{-self}::m$$

Proof: According to the Stripe projection for *self* axis, we have that for StripeSet $\mathcal{SS}_m$ the following holds:

$$\mathsf{ss\_nodes}(\mathcal{SS}_m) = \{\, x \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge \tau(x,m) \,\} \tag{A.73}$$

From Equations (A.68) , (A.73) we have that:

$$\forall z \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \exists x,y : x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge y \in \mathsf{parent}^*(x) \wedge \tau(y,n) \wedge z = y \wedge \tau(z,m) \tag{A.74}$$

Now, for any context node x, we have:

$$\begin{aligned}
\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/ancestor\text{-}or\text{-self}::m]\!](x,\mathcal{RS}_q) \\
&= \mathcal{V}(\mathcal{S}[\![path/ancestor\text{-}or\text{-self}::m]\!]x,\mathcal{RS}_q) \\
&= \{\, x_3 \,|\, x_3 \in \mathcal{S}[\![path/ancestor\text{-}or\text{-self}::m]\!]x \wedge x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q) \,\} \\
&= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![ancestor\text{-}or\text{-self}::m]\!]x_1 \wedge \\
&\qquad x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q) \,\} \\
&= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{parent}^*(x_1) \wedge \tau(x_3,m) \wedge \\
&\qquad x_3 \in \mathsf{ss_nodes}(\mathcal{SS}_m) \,\} \\
&\overset{(A.74)}{=} \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{parent}^*(x_1) \wedge \tau(x_3,m) \wedge \\
&\qquad x_1 \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{parent}^*(x_1) \wedge \tau(x_2,n) \wedge \\
&\qquad x_3 = x_2 \wedge \tau(x_3,m) \,\} \\
&\overset{(A.69)}{=} \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{parent}^*(x_1) \wedge \tau(x_2,n) \wedge \\
&\qquad x_3 = x_2 \wedge \tau(x_3,m) \,\} \\
&= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![ancestor\text{-}or\text{-self}::n]\!]x_1 \wedge \\
&\qquad x_3 \in \mathcal{S}[\![self::m]\!]x_2 \,\} \\
&= \mathcal{S}[\![path/ancestor\text{-}or\text{-self}::n/self::m]\!]x
\end{aligned}$$

□

Equivalence 2.30: For any path expression *path*, the following holds:

$$path/ancestor\text{-}or\text{-self}::n/parent::m \overset{\text{srx}}{\equiv} path/ancestor::m$$

Proof: According to the Stripe projection for *parent* axis, we have that for StripeSet $\mathcal{SS}_m$ the following holds:

$$\begin{aligned}\mathsf{ss\_nodes}(\mathcal{SS}_m) &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge \mathsf{path}(y) = \mathsf{maximal\_prefix}(\mathsf{path}(x)) \wedge \tau(y,m) \,\} \\ &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge y \in \mathsf{parent}(x) \wedge \tau(y,m) \,\} \qquad \text{(A.75)}\end{aligned}$$

From Equations (A.68) , (A.75) we have that:

$$\begin{aligned}\forall z \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \exists x,y : x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \\ \wedge\, y \in \mathsf{parent}^*(x) \wedge \tau(y,n) \wedge z \in \mathsf{parent}(y) \wedge \tau(z,m) \qquad \text{(A.76)}\end{aligned}$$

Now, for any context node x, we have:

$$\begin{aligned}\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/ancestor::m]\!](x,\mathcal{RS}_q) \\ &= \mathcal{V}(\mathcal{S}[\![path/ancestor::m]\!]x,\mathcal{RS}_q) \\ &= \{\, x_3 \,|\, x_3 \in \mathcal{S}[\![path/ancestor::m]\!]x \wedge x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q) \,\} \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![ancestor::m]\!]x_1 \wedge \\ &\qquad x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q) \,\} \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{parent}^+(x_1) \wedge \tau(x_3,m) \wedge \\ &\qquad x_3 \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \,\} \\ &\overset{\text{(A.76)}}{=} \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{parent}^+(x_1) \wedge \tau(x_3,m) \wedge \\ &\qquad x_1 \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{parent}^*(x_1) \wedge \tau(x_2,n) \wedge \\ &\qquad x_3 \in \mathsf{parent}(x_2) \wedge \tau(x_3,m) \,\} \\ &\overset{\text{(A.70)}}{=} \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{parent}^*(x_1) \wedge \tau(x_2,n) \wedge \\ &\qquad x_3 \in \mathsf{parent}(x_2) \wedge \tau(x_3,m) \,\} \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![ancestor\text{-}or\text{-self}::n]\!]x_1 \wedge \\ &\qquad x_3 \in \mathcal{S}[\![parent::m]\!]x_2 \,\} \\ &= \mathcal{S}[\![path/ancestor\text{-}or\text{-self}::n/parent::m]\!]x\end{aligned}$$

□

Equivalence 2.31: For any path expression *path*, the following holds:

$$path/ancestor\text{-}or\text{-self}::n/ancestor::m \overset{\text{srx}}{\equiv} path/ancestor::m$$

Proof: According to the Stripe projection for *ancestor* axis, we have that for StripeSet $\mathcal{SS}_m$ the following holds:

$$\begin{aligned}\mathsf{ss\_nodes}(\mathcal{SS}_m) &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge \mathsf{path}(y) \in \mathsf{proper\_prefix}(\mathsf{path}(x)) \wedge \tau(y,m) \,\} \\ &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge y \in \mathsf{parent}^+(x) \wedge \tau(y,m) \,\} \qquad \text{(A.77)}\end{aligned}$$

From Equations (A.68) , (A.77) we have that:

$$\begin{aligned}\forall z \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \exists x,y : x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \\ \wedge\, y \in \mathsf{parent}^*(x) \wedge \tau(y,n) \wedge z \in \mathsf{parent}^+(y) \wedge \tau(z,m) \qquad \text{(A.78)}\end{aligned}$$

Now, for any context node x, we have:

$$\begin{aligned}\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/ancestor::m]\!](x,\mathcal{RS}_q) \\ &= \mathcal{V}(\mathcal{S}[\![path/ancestor::m]\!]x, \mathcal{RS}_q) \\ &= \{\, x_3 \,|\, x_3 \in \mathcal{S}[\![path/ancestor::m]\!]x \wedge x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q) \,\} \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![ancestor::m]\!]x_1 \wedge \\ &\qquad x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q) \,\} \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{parent}^+(x_1) \wedge \tau(x_3,m) \wedge \\ &\qquad x_3 \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \,\} \\ &\overset{\text{(A.78)}}{=} \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{parent}^+(x_1) \wedge \tau(x_3,m) \wedge \\ &\qquad x_1 \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{parent}^*(x_1) \wedge \tau(x_2,n) \wedge \\ &\qquad x_3 \in \mathsf{parent}^+(x_2) \wedge \tau(x_3,m) \,\} \\ &\overset{\text{(A.71)}}{=} \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{parent}^*(x_1) \wedge \tau(x_2,n) \wedge \\ &\qquad x_3 \in \mathsf{parent}^+(x_2) \wedge \tau(x_3,m) \,\} \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![ancestor\text{-}or\text{-self}::n]\!]x_1 \wedge \\ &\qquad x_3 \in \mathcal{S}[\![ancestor::m]\!]x_2 \,\} \\ &= \mathcal{S}[\![path/ancestor\text{-}or\text{-self}::n/ancestor::m]\!]x\end{aligned}$$

□

Equivalence 2.32: For any path expression *path*, the following holds:

$$path/ancestor\text{-}or\text{-self}::n/ancestor\text{-}or\text{-self}::m \overset{\text{srx}}{\equiv} path/ancestor\text{-}or\text{-self}::m$$

Proof: According to the Stripe projection for *parent* axis, we have that for StripeSet $\mathcal{SS}_m$ the following holds:

$$\begin{aligned}\mathsf{ss\_nodes}(\mathcal{SS}_m) &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge \mathsf{path}(y) \in \mathsf{prefix}(\mathsf{path}(x)) \wedge \tau(y,m) \,\} \\ &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge y \in \mathsf{parent}^*(x) \wedge \tau(y,m) \,\} \end{aligned} \tag{A.79}$$

From Equations (A.68) , (A.79) we have that:

$$\begin{aligned}\forall z \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \exists x, y : x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \\ \wedge\, y \in \mathsf{parent}^*(x) \wedge \tau(y,n) \wedge z \in \mathsf{parent}^*(y) \wedge \tau(z,m)\end{aligned} \tag{A.80}$$

Now, for any context node x, we have:

$$\begin{aligned}\mathcal{F}[\![q']\!](x, \mathcal{RS}_q) &= \mathcal{F}[\![path/ancestor\text{-}or\text{-}\mathrm{self}::m]\!](x, \mathcal{RS}_q) \\ &= \mathcal{V}(\mathcal{S}[\![path/ancestor\text{-}or\text{-}\mathrm{self}::m]\!]x, \mathcal{RS}_q) \\ &= \{\, x_3 \,|\, x_3 \in \mathcal{S}[\![path/ancestor\text{-}or\text{-}\mathrm{self}::m]\!]x \wedge x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q) \,\} \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![ancestor\text{-}or\text{-}\mathrm{self}::m]\!]x_1 \wedge \\ &\quad\; x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q) \,\} \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{parent}^*(x_1) \wedge \tau(x_3,m) \wedge \\ &\quad\; x_3 \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \,\} \\ &\overset{(A.80)}{=} \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{parent}^*(x_1) \wedge \tau(x_3,m) \wedge \\ &\quad\; x_1 \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{parent}^*{+}(x_1) \wedge \tau(x_2,n) \wedge \\ &\quad\; x_3 \in \mathsf{parent}^*(x_2) \wedge \tau(x_3,m) \,\} \\ &\overset{(A.72)}{=} \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{parent}^*(x_1) \wedge \tau(x_2,n) \wedge \\ &\quad\; x_3 \in \mathsf{parent}^*(x_2) \wedge \tau(x_3,m) \,\} \\ &= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![ancestor\text{-}or\text{-}\mathrm{self}::n]\!]x_1 \wedge \\ &\quad\; x_3 \in \mathcal{S}[\![ancestor\text{-}or\text{-}\mathrm{self}::m]\!]x_2 \,\} \\ &= \mathcal{S}[\![path/ancestor\text{-}or\text{-}\mathrm{self}::n/ancestor\text{-}or\text{-}\mathrm{self}::m]\!]x \end{aligned}$$

□

A.7 Self Axis

We conclude with the *self* axis equivalence rules. The proofs resemble the ones that involve a *self* axis in $step_2$ expression of query $q \in Q$. We provide the proof for equivalence 2.4. The rest can be proven in a similar way.

Equivalence 2.4: For any path expression *path*, the following holds:

$$path/self::n/child::m \overset{\text{srx}}{\equiv} path/child::m$$

Proof: According to Stripe projection for *self* and *child* axis, we have that:

$$\mathsf{ss\_nodes}(\mathcal{SS}_n) = \{\, x \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge \tau(x,n) \,\} \tag{A.81}$$

$$\begin{aligned}\mathsf{ss\_nodes}(\mathcal{SS}_m) &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge \mathsf{path}(x) = \mathsf{maximal\_prefix}(\mathsf{path}(y)) \wedge \tau(y,m) \,\} \\ &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge y \in \mathsf{children}(x) \wedge \tau(y,m) \,\} \end{aligned} \tag{A.82}$$

From Equations (A.81) , (A.82) we have that:

$$\begin{aligned}\forall z \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \exists x,y : x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \\ \wedge y = x \wedge \tau(y,n) \wedge z \in \mathsf{children}(y) \wedge \tau(z,m)\end{aligned} \tag{A.83}$$

Now, for any context node x, we have:

$$\begin{aligned}
\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/child::m]\!](x,\mathcal{RS}_q) \\
&= \mathcal{V}(\mathcal{S}[\![path/child::m]\!]x,\mathcal{RS}_q) \\
&= \{\, x_3 \,|\, x_3 \in \mathcal{S}[\![path/child::m]\!]x \wedge x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q) \,\} \\
&= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![child::m]\!]x_1 \wedge \\
&\quad\; x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q) \,\} \\
&= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{children}(x_1) \wedge \tau(x_3,m) \wedge \\
&\quad\; x_3 \in \mathsf{ss_nodes}(\mathcal{SS}_m) \,\} \\
&\overset{(A.83)}{=} \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{children}(x_1) \wedge \tau(x_3,m) \wedge \\
&\quad\; x_1 \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge x_2 = x_1 \wedge \tau(x_2,n) \wedge \\
&\quad\; x_3 \in \mathsf{children}(x_2) \wedge \tau(x_3,m) \,\} \\
&= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 = x_1 \wedge \tau(x_2,n) \wedge \\
&\quad\; x_3 \in \mathsf{children}(x_2) \wedge \tau(x_3,m) \,\} \\
&= \{\, x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![self::n]\!]x_1 \wedge x_3 \in \mathcal{S}[\![child::m]\!]x_2 \,\} \\
&= \mathcal{S}[\![path/self::n/child::m]\!]x
\end{aligned}$$

□

A.8 Following Axis

We remind the reader that the condition c for which the following axis-related equivalences hold is defined as:

$$\forall x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path})\ \nexists y : y \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge y \in \mathsf{parent}^+(x) \tag{A.84}$$

According to the Stripe projection for *following* axis and condition c, we have that for StripeSet $\mathcal{SS}_n$ the following holds:

$$\begin{aligned}\mathsf{ss\_nodes}(\mathcal{SS}_n) &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge y \in \mathsf{children}^+(\mathsf{root}(x)) \wedge \tau(y,n)\,\} \\ &\overset{(A.84)}{=} \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge y \in \mathsf{children}^+(\mathsf{root}(x)) \wedge \\ &\qquad y \notin \mathsf{parent}^+(x) \wedge \tau(y,n)\,\}\end{aligned} \tag{A.85}$$

Equivalence 2.42: For any path expression *path*, the following holds:

$$path/following::n/child::m \underset{c}{\overset{\mathsf{srx}}{\equiv}} path/following::m$$

Proof: According to the Stripe projection for *child* axis, we have that:

$$\begin{aligned}\mathsf{ss\_nodes}(\mathcal{SS}_m) &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge \mathsf{path}(x) = \mathsf{maximal\_prefix}(\mathsf{path}(y)) \wedge \tau(y,m)\,\} \\ &= \{\, y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge y \in \mathsf{children}(x) \wedge \tau(y,m)\,\}\end{aligned} \tag{A.86}$$

From Equations (A.85) , (A.86) we have that:

$$\begin{aligned}&\forall z \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \exists x,y : x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge \\ &y \in \mathsf{children}^+(\mathsf{root}(x)) \wedge \tau(y,n) \wedge y \notin \mathsf{parent}^+(x) \wedge z \in \mathsf{children}(y) \wedge \tau(z,m)\end{aligned} \tag{A.87}$$

$$
\begin{aligned}
\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/following::m]\!](x,\mathcal{RS}_q)\\
&= \mathcal{V}(\mathcal{S}[\![path/following::m]\!]x,\mathcal{RS}_q)\\
&= \{x_3 \mid x_3 \in \mathcal{S}[\![path/following::m]\!]x \wedge x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![following::m]\!]x_1 \wedge\\
&\quad x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{following}(x_1) \wedge \tau(x_3,m) \wedge\\
&\quad x_3 \in \mathsf{ss\_nodes}(\mathcal{SS}_m)\}\\
&\overset{(A.87)}{=} \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{following}(x_1) \wedge \tau(x_3,m) \wedge\\
&\quad x_1 \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}^+(\mathsf{root}(x_1)) \wedge \tau(x_2,n) \wedge\\
&\quad x_2 \notin \mathsf{parent}^+(x_1) \wedge x_3 \in \mathsf{children}(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge\\
&\quad (((x_2 \in \mathsf{following}(x_1) \vee x_2 \in \mathsf{parent}^+(x_1)) \wedge \tau(x_2,*)) \wedge\\
&\quad x_3 \in \mathsf{children}(x_2) \wedge \tau(x_3,m)) \wedge\\
&\quad x_1 \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}^+(\mathsf{root}(x_1)) \wedge \tau(x_2,n) \wedge\\
&\quad x_2 \notin \mathsf{parent}^+(x_1) \wedge x_3 \in \mathsf{children}(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{following}(x_1) \wedge \tau(x_2,n) \wedge\\
&\quad x_3 \in \mathsf{children}(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![following::n]\!]x_1 \wedge\\
&\quad x_3 \in \mathcal{S}[\![child::m]\!]x_2\}\\
&= \mathcal{S}[\![path/following::n/child::m]\!]x
\end{aligned}
$$

□

Equivalence 2.43: For any path expression *path*, the following holds:

$$path/following::n/descendant::m \overset{\mathsf{srx}}{\underset{c}{\equiv}} path/following::m$$

Proof: According to the Stripe projection for *descendant* axis, we have that:

$$
\begin{aligned}
\mathsf{ss\_nodes}(\mathcal{SS}_m) &= \{y \mid x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge \mathsf{path}(x) \in \mathsf{proper\_prefix}(\mathsf{path}(y)) \wedge \tau(y,m)\}\\
&= \{y \mid x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge y \in \mathsf{children}^+(x) \wedge \tau(y,m)\} \qquad \text{(A.88)}
\end{aligned}
$$

From Equations (A.85) , (A.88) we have that:

$$
\begin{aligned}
&\forall z \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \exists x,y : x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge\\
&y \in \mathsf{children}^+(\mathsf{root}(x)) \wedge \tau(y,n) \wedge y \notin \mathsf{parent}^+(x) \wedge z \in \mathsf{children}^+(y) \wedge \tau(z,m) \qquad \text{(A.89)}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/following{::}m]\!](x,\mathcal{RS}_q)\\
&= \mathcal{V}(\mathcal{S}[\![path/following{::}m]\!]x,\mathcal{RS}_q)\\
&= \{x_3 \,|\, x_3 \in \mathcal{S}[\![path/following{::}m]\!]x \wedge x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![following{::}m]\!]x_1 \wedge\\
&\quad x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{following}(x_1) \wedge \tau(x_3,m) \wedge\\
&\quad x_3 \in \mathsf{ss\_nodes}(\mathcal{SS}_m)\}\\
&\overset{(A.89)}{=} \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{following}(x_1) \wedge \tau(x_3,m) \wedge\\
&\quad x_1 \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}^+(\mathsf{root}(x_1)) \wedge \tau(x_2,n) \wedge\\
&\quad x_2 \notin \mathsf{parent}^+(x_1) \wedge x_3 \in \mathsf{children}^+(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge\\
&\quad (((x_2 \in \mathsf{following}(x_1) \vee x_2 \in \mathsf{parent}^+(x_1)) \wedge \tau(x_2,*)) \wedge\\
&\quad x_3 \in \mathsf{children}^+(x_2) \wedge \tau(x_3,m)) \wedge\\
&\quad x_1 \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}^+(\mathsf{root}(x_1)) \wedge \tau(x_2,n) \wedge\\
&\quad x_2 \notin \mathsf{parent}^+(x_1) \wedge x_3 \in \mathsf{children}^+(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{following}(x_1) \wedge \tau(x_2,n) \wedge\\
&\quad x_3 \in \mathsf{children}^+(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![following{::}n]\!]x_1 \wedge\\
&\quad x_3 \in \mathcal{S}[\![descendant{::}m]\!]x_2\}\\
&= \mathcal{S}[\![path/following{::}n/descendant{::}m]\!]x
\end{aligned}
$$

□

Equivalence 2.44: For any path expression *path*, the following holds:

$$path/following{::}n/descendant\text{-}or\text{-}self{::}m \underset{c}{\overset{\mathsf{srx}}{\equiv}} path/following{::}m$$

Proof: According to the Stripe projection for *descendant-or-self* axis, we have that:

$$
\begin{aligned}
\mathsf{ss\_nodes}(\mathcal{SS}_m) &= \{y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge \mathsf{path}(x) \in \mathsf{prefix}(\mathsf{path}(y)) \wedge \tau(y,m)\}\\
&= \{y \,|\, x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge y \in \mathsf{children}^*(x) \wedge \tau(y,m)\} \qquad (A.90)
\end{aligned}
$$

From Equations (A.85) , (A.90) we have that:

$$
\begin{aligned}
&\forall z \in \mathsf{ss\_nodes}(\mathcal{SS}_m) \exists x,y : x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge\\
&y \in \mathsf{children}^+(\mathsf{root}(x)) \wedge \tau(y,n) \wedge y \notin \mathsf{parent}^+(x) \wedge z \in \mathsf{children}^*(y) \wedge \tau(z,m) \qquad (A.91)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/following::m]\!](x,\mathcal{RS}_q)\\
&= \mathcal{V}(\mathcal{S}[\![path/following::m]\!]x,\mathcal{RS}_q)\\
&= \{x_3 \mid x_3 \in \mathcal{S}[\![path/following::m]\!]x \wedge x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![following::m]\!]x_1 \wedge\\
&\quad x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{following}(x_1) \wedge \tau(x_3,m) \wedge\\
&\quad x_3 \in \mathsf{ss\_nodes}(\mathcal{SS}_m)\}\\
&\overset{(A.91)}{=} \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{following}(x_1) \wedge \tau(x_3,m) \wedge\\
&\quad x_1 \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}^+(\mathsf{root}(x_1)) \wedge \tau(x_2,n) \wedge\\
&\quad x_2 \notin \mathsf{parent}^+(x_1) \wedge x_3 \in \mathsf{children}^*(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge\\
&\quad (((x_2 \in \mathsf{following}(x_1) \vee x_2 \in \mathsf{parent}^+(x_1)) \wedge \tau(x_2,*)) \wedge\\
&\quad x_3 \in \mathsf{children}^*(x_2) \wedge \tau(x_3,m)) \wedge\\
&\quad x_1 \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}^+(\mathsf{root}(x_1)) \wedge \tau(x_2,n) \wedge\\
&\quad x_2 \notin \mathsf{parent}^+(x_1) \wedge x_3 \in \mathsf{children}^*(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{following}(x_1) \wedge \tau(x_2,n) \wedge\\
&\quad x_3 \in \mathsf{children}^*(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![following::n]\!]x_1 \wedge\\
&\quad x_3 \in \mathcal{S}[\![descendant\text{-}or\text{-}self::m]\!]x_2\}\\
&= \mathcal{S}[\![path/following::n/descendant\text{-}or\text{-}self::m]\!]x
\end{aligned}
$$

□

Equivalence 2.41: For any path expression *path*, the following holds:

$$path/following::n/self::m \overset{\mathsf{srx}}{\equiv} path/following::m$$

Proof: Note that the this is not a conditional equivalence. Thus, from Stripe projection for *following* and *self* and axis, we have that for StripeSets $\mathcal{SS}_n$, $\mathcal{SS}_m$, the following hold:

$$
\begin{aligned}
\mathsf{ss\_nodes}(\mathcal{SS}_n) &= \{y \mid x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge\\
&\quad y \in \mathsf{children}^+(\mathsf{root}(x)) \wedge \tau(y,n)\}\\
\mathsf{ss\_nodes}(\mathcal{SS}_m) &= \{y \mid x \in \mathsf{ss\_nodes}(\mathcal{SS}_n) \wedge y = x \wedge \tau(y,m)\}
\end{aligned}
$$

and thus:

$$\forall z \in \mathsf{ss\_nodes}(\mathcal{SS}_m)\exists x,y : x \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge \\ y \in \mathsf{children}^+(\mathsf{root}(x)) \wedge \tau(y,n) \wedge y = x \wedge \tau(z,m) \quad \text{(A.92)}$$

$$\begin{aligned}
\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/following::m]\!](x,\mathcal{RS}_q) \\
&= \mathcal{V}(\mathcal{S}[\![path/following::m]\!]x,\mathcal{RS}_q) \\
&= \{x_3 \,|\, x_3 \in \mathcal{S}[\![path/following::m]\!]x \wedge x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q)\} \\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![following::m]\!]x_1 \wedge \\
&\quad\; x_3 \in \mathsf{ss_nodes}(\mathcal{RS}_q)\} \\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{following}(x_1) \wedge \tau(x_3,m) \wedge \\
&\quad\; x_3 \in \mathsf{ss_nodes}(\mathcal{SS}_m)\} \\
&\overset{\text{(A.92)}}{=} \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{following}(x_1) \wedge \tau(x_3,m) \wedge \\
&\quad\; x_1 \in \mathsf{ss_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}^+(\mathsf{root}(x_1)) \wedge \tau(x_2,n) \wedge \\
&\quad\; x_3 = x_2 \wedge \tau(x_3,m)\} \\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{following}(x_1) \wedge \tau(x_2,n) \wedge \\
&\quad\; x_3 \in \mathsf{self}(x_2) \wedge \tau(x_3,m)\} \\
&= \{x_3 \,|\, x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![following::n]\!]x_1 \wedge \\
&\quad\; x_3 \in \mathcal{S}[\![self::m]\!]x_2\} \\
&= \mathcal{S}[\![path/following::n/self::m]\!]x
\end{aligned}$$

A.9 Preceding Axis

Since the Stripe projection process for *preceding* axis has the same result as for *following* axis, the findings for StripeSets $\mathcal{SS}_n$, $\mathcal{SS}_m$ are considered the same as in the set of following axis-related equivalences.

Equivalence 2.46: For any path expression *path*, the following holds:

$$path/preceding::n/child::m \overset{\mathsf{srx}}{\underset{c}{\equiv}} path/preceding::m$$

Proof:

$$
\begin{aligned}
\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/preceding{::}m]\!](x,\mathcal{RS}_q)\\
&= \mathcal{V}(\mathcal{S}[\![path/preceding{::}m]\!]x,\mathcal{RS}_q)\\
&= \{x_3 \mid x_3 \in \mathcal{S}[\![path/preceding{::}m]\!]x \wedge x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![preceding{::}m]\!]x_1 \wedge\\
&\quad x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{preceding}(x_1) \wedge \tau(x_3,m) \wedge\\
&\quad x_3 \in \mathsf{ss\_nodes}(\mathcal{SS}_m)\}\\
&\overset{(A.87)}{=} \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{preceding}(x_1) \wedge \tau(x_3,m) \wedge\\
&\quad x_1 \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}^+(\mathsf{root}(x_1)) \wedge \tau(x_2,n) \wedge\\
&\quad x_2 \notin \mathsf{parent}^+(x_1) \wedge x_3 \in \mathsf{children}(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge\\
&\quad (((x_2 \in \mathsf{preceding}(x_1) \vee x_2 \in \mathsf{parent}^+(x_1)) \wedge \tau(x_2,*)) \wedge\\
&\quad x_3 \in \mathsf{children}(x_2) \wedge \tau(x_3,m)) \wedge\\
&\quad x_1 \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}^+(\mathsf{root}(x_1)) \wedge \tau(x_2,n) \wedge\\
&\quad x_2 \notin \mathsf{parent}^+(x_1) \wedge x_3 \in \mathsf{children}(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{preceding}(x_1) \wedge \tau(x_2,n) \wedge\\
&\quad x_3 \in \mathsf{children}(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![preceding{::}n]\!]x_1 \wedge\\
&\quad x_3 \in \mathcal{S}[\![child{::}m]\!]x_2\}\\
&= \mathcal{S}[\![path/preceding{::}n/child{::}m]\!]x
\end{aligned}
$$

□

Equivalence 2.47: For any path expression *path*, the following holds:

$$path/preceding{::}n/descendant{::}m \underset{c}{\overset{\mathsf{srx}}{\equiv}} path/preceding{::}m$$

Proof:

$$
\begin{aligned}
\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/preceding{::}m]\!](x,\mathcal{RS}_q)\\
&= \mathcal{V}(\mathcal{S}[\![path/preceding{::}m]\!]x,\mathcal{RS}_q)\\
&= \{x_3 \mid x_3 \in \mathcal{S}[\![path/preceding{::}m]\!]x \wedge x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![preceding{::}m]\!]x_1 \wedge\\
&\quad x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{preceding}(x_1) \wedge \tau(x_3,m) \wedge\\
&\quad x_3 \in \mathsf{ss\_nodes}(\mathcal{SS}_m)\}\\
&\overset{(A.89)}{=} \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{preceding}(x_1) \wedge \tau(x_3,m) \wedge\\
&\quad x_1 \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}^+(\mathsf{root}(x_1)) \wedge \tau(x_2,n) \wedge\\
&\quad x_2 \notin \mathsf{parent}^+(x_1) \wedge x_3 \in \mathsf{children}^+(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge\\
&\quad (((x_2 \in \mathsf{preceding}(x_1) \vee x_2 \in \mathsf{parent}^+(x_1)) \wedge \tau(x_2,*)) \wedge\\
&\quad x_3 \in \mathsf{children}^+(x_2) \wedge \tau(x_3,m)) \wedge\\
&\quad x_1 \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}^+(\mathsf{root}(x_1)) \wedge \tau(x_2,n) \wedge\\
&\quad x_2 \notin \mathsf{parent}^+(x_1) \wedge x_3 \in \mathsf{children}^+(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{preceding}(x_1) \wedge \tau(x_2,n) \wedge\\
&\quad x_3 \in \mathsf{children}^+(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![preceding{::}n]\!]x_1 \wedge\\
&\quad x_3 \in \mathcal{S}[\![descendant{::}m]\!]x_2\}\\
&= \mathcal{S}[\![path/preceding{::}n/descendant{::}m]\!]x
\end{aligned}
$$

□

Equivalence 2.48:

$$
\begin{aligned}
\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/preceding{::}m]\!](x,\mathcal{RS}_q)\\
&= \mathcal{V}(\mathcal{S}[\![path/preceding{::}m]\!]x,\mathcal{RS}_q)\\
&= \{x_3 \mid x_3 \in \mathcal{S}[\![path/preceding{::}m]\!]x \wedge x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![preceding{::}m]\!]x_1 \wedge\\
&\quad x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{preceding}(x_1) \wedge \tau(x_3,m) \wedge\\
&\quad x_3 \in \mathsf{ss\_nodes}(\mathcal{SS}_m)\}\\
&\overset{(A.91)}{=} \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{preceding}(x_1) \wedge \tau(x_3,m) \wedge\\
&\quad x_1 \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}^+(\mathsf{root}(x_1)) \wedge \tau(x_2,n) \wedge\\
&\quad x_2 \notin \mathsf{parent}^+(x_1) \wedge x_3 \in \mathsf{children}^*(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge\\
&\quad (((x_2 \in \mathsf{preceding}(x_1) \vee x_2 \in \mathsf{parent}^+(x_1)) \wedge \tau(x_2,*)) \wedge\\
&\quad x_3 \in \mathsf{children}^*(x_2) \wedge \tau(x_3,m)) \wedge\\
&\quad x_1 \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}^+(\mathsf{root}(x_1)) \wedge \tau(x_2,n) \wedge\\
&\quad x_2 \notin \mathsf{parent}^+(x_1) \wedge x_3 \in \mathsf{children}^*(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{preceding}(x_1) \wedge \tau(x_2,n) \wedge\\
&\quad x_3 \in \mathsf{children}^*(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![preceding{::}n]\!]x_1 \wedge\\
&\quad x_3 \in \mathcal{S}[\![descendant\text{-}or\text{-}self{::}m]\!]x_2\}\\
&= \mathcal{S}[\![path/preceding{::}n/descendant\text{-}or\text{-}self{::}m]\!]x
\end{aligned}
$$

□

Equivalence 2.45: For any path expression *path*, the following holds:

$$path/preceding{::}n/self{::}m \overset{\mathsf{srx}}{\equiv} path/preceding{::}m$$

Proof:

$$
\begin{aligned}
\mathcal{F}[\![q']\!](x,\mathcal{RS}_q) &= \mathcal{F}[\![path/preceding{::}m]\!](x,\mathcal{RS}_q)\\
&= \mathcal{V}(\mathcal{S}[\![path/preceding{::}m]\!]x,\mathcal{RS}_q)\\
&= \{x_3 \mid x_3 \in \mathcal{S}[\![path/preceding{::}m]\!]x \wedge x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathcal{S}[\![preceding{::}m]\!]x_1 \wedge\\
&\quad\ x_3 \in \mathsf{ss\_nodes}(\mathcal{RS}_q)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{preceding}(x_1) \wedge \tau(x_3,m) \wedge\\
&\quad\ x_3 \in \mathsf{ss\_nodes}(\mathcal{SS}_m)\}\\
&\overset{(A.92)}{=} \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_3 \in \mathsf{preceding}(x_1) \wedge \tau(x_3,m) \wedge\\
&\quad\ x_1 \in \mathsf{ss\_nodes}(\mathcal{SS}_{path}) \wedge x_2 \in \mathsf{children}^+(\mathsf{root}(x_1)) \wedge \tau(x_2,n) \wedge\\
&\quad\ x_3 = x_2 \wedge \tau(x_3,m)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathsf{preceding}(x_1) \wedge \tau(x_2,n) \wedge\\
&\quad\ x_3 \in \mathsf{self}(x_2) \wedge \tau(x_3,m)\}\\
&= \{x_3 \mid x_1 \in \mathcal{S}[\![path]\!]x \wedge x_2 \in \mathcal{S}[\![preceding{::}n]\!]x_1 \wedge\\
&\quad\ x_3 \in \mathcal{S}[\![self{::}m]\!]x_2\}\\
&= \mathcal{S}[\![path/preceding{::}n/self{::}m]\!]x
\end{aligned}
$$

Appendix B

Query Testbed

a1	`/site/regions/*/item`
a2	`/site/closed_auctions/closed_auction/annotation/description/parlist/listitem/parlist/listitem/text/keyword/emph/text()`
a3	`/site/regions/asia/item[shipping]/description`
a4	`/site/closed_auctions/closed_auction[annotation/description/parlist/listitem/parlist/listitem/text/keyword/emph/text()]/seller/@person`
a5	`/site/people/person[@id = 'person0']/name/text()`
a6	`//person[profile/@income]/name`
b1	`/descendant::open_auction/descendant::description`
b2	`/descendant::open_auction/descendant::description/descendant::listitem`
b3	`/descendant::open_auction/descendant::description/descendant::listitem/descendant::keyword`
b4	`/site/closed_auctions/closed_auction[descendant::keyword]/date`
c1	`/site/open_auctions/open_auction/bidder[personref/@person='person1']/following-sibling::bidder/personref/@person`
c2	`/site/open_auctions/open_auction/bidder[following-sibling::bidder]`
c3	`/site/open_auctions/open_auction[bidder[personref/@person='person1']/following-sibling::bidder[personref/@person='person2409']]`
d1	`/site/regions/*/item[@id='item2000']/following::item`
d2	`/site/regions/*/item[following::item]/name`
e1	`/site/people/person[profile/gender and profile/age]/name`
e2	`/site/people/person[phone or homepage]/name`
e3	`/site/people/person[address and (phone or homepage) and (creditcard or profile)]/name`
e4	`/site/people/person[not(homepage)]`
e5	`/site/open_auctions/open_auction[bidder and not(bidder/following-sibling::bidder)]/interval`

Table B.1: Xmark query testbed

a1	//eNest//eOccasional/@aRef
a2	//eNest/eOccasional/@aRef
b1	//eNest[@aString = 'Sing a song of oneB4']/@aUnique1
b2	//eNest[@aSixtyFour=2]/@aUnique1
b3	//eNest[@aString = 'Sing a song of oneB1']/@aUnique1
c1	//eNest[@aLevel=13][./eNest[@aSixteen=3]]/@aUnique1
c2	//eNest[@aLevel=15][./eNest[@aSixtyFour=3]]/@aUnique1
c3	//eNest[@aLevel=11][./eNest[@aFour=3]]/@aUnique1
d1	//eNest[@aLevel=13][.//eNest[@aSixteen=3]]/@aUnique1
d2	//eNest[@aFour=3][.//eNest[@aSixtyFour=3]]/@aUnique1
d3	//eNest[@aSixtyFour=9][.//eNest[@aFour=3]]/@aUnique1
e1	//eNest[@aLevel=11][./eNest[@aFour=3]][./eNest[@aSixtyFour=3]]/@aUnique1
e2	//eNest[@aFour=1][./eNest[@aLevel=11]][./eNest[@aSixtyFour=3]]/@aUnique1
e3	//eNest[@aFour=1][./eNest[@aLevel=11]][.//eNest[@aSixtyFour=3]]/@aUnique1
e4	//eNest[@aLevel=11][.//eNest[@aFour=3]][.//eNest[@aSixtyFour=3]]/@aUnique1
e5	//eNest[@aFour=1][.//eNest[@aLevel=11]][.//eNest[@aSixtyFour=3]]/@aUnique1
f1	//eNest[@aFour=3][./eNest[@aSixteen=3]/ eNest[@aSixteen=5]/eNest[@aLevel=16]]/@aUnique1
f2	//eNest[@aFour=3][.//eNest[@aSixteen=3]// eNest[@aSixteen=5]//eNest[@aLevel=16]]/@aUnique1

Table B.2: Mbench query testbed (a)

g1	//eNest/following::eOccasional/@aRef
g2	//eNest[@aLevel=13]/following::eNest[@aSixteen=3]/@aUnique1
g3	//eNest[@aLevel=11]/following::eNest[@aFour=3]/ eNest[@aSixtyFour=3]/@aUnique1
h1	//eNest[@aLevel=13][./following::eNest[@aSixteen=3]]/@aUnique1
h2	//eNest[@aLevel=15][./following::eNest[@aSixtyFour=3]]/@aUnique1
h3	//eNest[@aLevel=11][./following::eNest[@aFour=3]]/@aUnique1
h4	//eNest[@aLevel=11][./eNest[@aFour=3]] [./following::eNest[@aSixtyFour=3]]/@aUnique1
h5	//eNest[@aLevel=11][./following::eNest[@aFour=3]] [./eNest[@aSixtyFour=3]]/@aUnique1
h6	//eNest[@aLevel=13][./following::eNest[@aFour=3]] [./following::eNest[@aSixtyFour=3]]/@aUnique1
h7	//eNest[@aFour=3][./eNest[@aSixteen=3]/ following::eNest[@aSixteen=5]/eNest[@aLevel=16]]/@aUnique1
i1	//eNest/following-sibling::eOccasional/@aRef
i2	//eNest[@aLevel=13]/following-sibling::eNest[@aSixteen=3]/@aUnique1
i3	//eNest[@aLevel=11]/following-sibling::eNest[@aFour=3]/ eNest[@aSixtyFour=3]/@aUnique1
j1	//eNest[@aLevel=13][./following-sibling::eNest[@aSixteen=3]]/@aUnique1
j2	//eNest[@aLevel=15][./following-sibling::eNest[@aSixtyFour=3]]/@aUnique1
j3	//eNest[@aLevel=11][./following-sibling::eNest[@aFour=3]]/@aUnique1
j4	//eNest[@aLevel=11][./eNest[@aFour=3]] [./following-sibling::eNest[@aSixtyFour=3]]/@aUnique1
j5	//eNest[@aLevel=11][./following-sibling::eNest[@aFour=3]] [./eNest[@aSixtyFour=3]]/@aUnique1
j6	//eNest[@aLevel=13][./following-sibling::eNest[@aFour=3]] [./following-sibling::eNest[@aSixtyFour=3]]/@aUnique1
j7	//eNest[@aFour=3][./eNest[@aSixteen=3]/ following-sibling::eNest[@aSixteen=5]/eNest[@aLevel=16]]/@aUnique1

Table B.3: Mbench query testbed (b)

a1	/db/dblp/article/title
a2	/db/dblp/*/title
a3	/db/dblp/article/title | /db/dblp/inproceedings/title
a4	/db/dblp/book/series
a5	/db/dblp/descendant::phdthesis
a6	//article/@rating
a7	//www[./editor]/url
b1	/db/dblp/article[month="February"]
b2	/db/dblp/inproceedings[@key="conf/er/LockemannM91"]/title
b3	//inproceedings[./title = "Semantic Analysis Patterns."]/author
b4	/db/dblp/inproceedings[author="Christos H. Papadimitriou"]/title
b5	//inproceedings[./author = "Jim Gray"][./year = 1990]/@key
b6	/db/dblp/article[year=1991]/@key | /db/dblp/inproceedings[year=1991]/@key
c1	//*[./editor]/title
c2	/db/dblp/*[author="David J. DeWitt"]/@key
c3	//inproceedings[./* = "Semantic Analysis Patterns."]/author
c4	/db/dblp/phdthesis[* = "1996"]/author
c5	//*[@key="conf/er/LockemannM91"]/booktitle
c6	//*[booktitle="Object-Oriented Concepts, Databases, and Applications"]/title/text()
d1	//phdthesis[author = "Limsoon Wong"]/following::title
d2	//incollection/following::volume[./text()="2"]
e1	//phdthesis[author = "Goetz Graefe"]/following-sibling::phdthesis/title
e2	//proceedings[editor = "Peter Buneman"] [./following-sibling::proceedings[./year = 2002]]/title

Table B.4: DBLP query testbed

Bibliography

[1] A Quick Benchmark: Gzip vs. Bzip2 vs. LZMA. http://tukaani.org/lzma/benchmarks.

[2] Berkeley DB: Oracle Embeded Database. http://www.oracle.com/technology/products/berkeley-db/db/index.html.

[3] Berkeley DB Reference Guide. http://www.oracle.com/technology/documentation/berkeley-db/db/ref/toc.html.

[4] SAX, Simple API for XML. http://www.saxproject.org.

[5] The world wide web consortium (w3c).

[6] S. Abiteboul. Querying semi-structured data. In *ICDT*, pages 1–18, 1997.

[7] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.

[8] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient xml query pattern matching. In *ICDE*, pages 141–, 2002.

[9] A. Arion, A. Bonifati, G. Costa, S. D'Aguanno, I. Manolescu, and A. Pugliese. Efficient query evaluation over compressed xml data. In *EDBT*, pages 200–218, 2004.

[10] A. Arion, A. Bonifati, I. Manolescu, and A. Pugliese. Xquec: A query-conscious compressed xml database. *ACM Trans. Internet Techn.*, 7(2), 2007.

[11] A. Arion, A. Bonifati, I. Manolescu, and A. Pugliese. Path summaries and path partitioning in modern xml databases. *World Wide Web*, 11(1):117–151, 2008.

[12] D. S. Batory. On searching transposed files. *ACM Trans. Database Syst.*, 4(4):531–544, 1979.

[13] M. Benedikt, C. Y. Chan, W. Fan, R. Rastogi, S. Zheng, and A. Zhou. Dtd-directed publishing with attribute translation grammars. In *VLDB*, pages 838–849, 2002.

[14] S. Boag, A. Berglund, D. Chamberlin, J. Siméon, M. Kay, J. Robie, and M. F. Fernández. XML path language (XPath) 2.0. W3C recommendation, W3C, Jan. 2007. http://www.w3.org/TR/2007/REC-xpath20-20070123/.

[15] P. Bohannon, J. Freire, P. Roy, and J. Siméon. From xml schema to relations: A cost-based approach to xml storage. In *ICDE*, pages 64–, 2002.

[16] P. A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. Ph.d. thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, May 2002.

[17] P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. Monetdb/xquery: a fast xquery processor powered by a relational engine. In *SIGMOD Conference*, pages 479–490, 2006.

[18] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal xml pattern matching. In *SIGMOD Conference*, pages 310–321, 2002.

[19] P. Buneman, B. Choi, W. Fan, R. Hutchison, R. Mann, and S. Viglas. Vectorizing and querying large xml repositories. In *ICDE*, pages 261–272, 2005.

[20] P. Buneman, M. Grohe, and C. Koch. Path queries on compressed xml. In *VLDB*, pages 141–152, 2003.

[21] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, Digital SRC Research Report, 1994.

[22] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. In *SIGMOD Conference*, pages 383–394, 1994.

[23] S. Chen, H.-G. Li, J. Tatemura, W.-P. Hsiung, D. Agrawal, and K. S. Candan. Twig^{2}stack: Bottom-up processing of generalized-tree-pattern queries over xml documents. In *VLDB*, pages 283–294, 2006.

[24] T. Chen, T. W. Ling, and C. Y. Chan. Prefix path streaming: A new clustering method for optimal holistic xml twig pattern matching. In *DEXA*, pages 801–810, 2004.

[25] T. Chen, J. Lu, and T. W. Ling. On boosting holism in xml twig pattern matching using structural indexing techniques. In *SIGMOD Conference*, pages 455–466, 2005.

[26] Z. Chen, H. V. Jagadish, L. V. S. Lakshmanan, and S. Paparizos. From tree patterns to generalized tree patterns: On efficient evaluation of xquery. In *VLDB*, pages 237–248, 2003.

[27] J. Cheng and W. Ng. Xqzip: Querying compressed xml using structural indexing. In *EDBT*, pages 219–236, 2004.

[28] S.-Y. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed xml documents. In *VLDB*, pages 263–274, 2002.

[29] B. Choi. What are real dtds like? In *WebDB*, pages 43–48, 2002.

[30] B. Choi, M. Mahoui, and D. Wood. On the optimality of holistic algorithms for twig queries. In *DEXA*, pages 28–37, 2003.

[31] G. P. Copeland and S. Khoshafian. A decomposition storage model. In *SIGMOD Conference*, pages 268–279, 1985.

[32] S. DeRose and J. Clark. XML path language (XPath) version 1.0. W3C recommendation, W3C, Nov. 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

[33] A. Deutsch, M. F. Fernández, and D. Suciu. Storing semistructured data with stored. In *SIGMOD Conference*, pages 431–442, 1999.

[34] P. Deutsch. DEFLATE compressed data format specification version 1.3, 1996.

[35] P. F. Dietz. Maintaining order in a linked list. In *STOC*, pages 122–127, 1982.

[36] W. Fan and L. Ma. Selectively storing xml data in relations. In S. Bressan, J. Küng, and R. Wagner, editors, *DEXA*, volume 4080 of *Lecture Notes in Computer Science*, pages 22–32. Springer, 2006.

[37] M. F. Fernandez, Y. Kadiyska, D. Suciu, A. Morishima, and W. C. Tan. Silkroute: A framework for publishing relational data in xml. *ACM Trans. Database Syst.*, 27(4):438–493, 2002.

[38] M. F. Fernández, J. Siméon, B. Choi, A. Marian, and G. Sur. Implementing xquery 1.0: The galax experience. In *VLDB*, pages 1077–1080, 2003.

[39] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a native xml base management system. *VLDB J.*, 11(4):292–314, 2002.

[40] D. Florescu and D. Kossmann. Storing and querying xml data using an rdmbs. *IEEE Data Eng. Bull.*, 22(3):27–34, 1999.

[41] J. Freire, J. R. Haritsa, M. Ramanath, P. Roy, and J. Siméon. Statix: making xml count. In *SIGMOD Conference*, pages 181–191, 2002.

[42] J. Gailly and M. Adler. gzip compressor. `http://www.gzip.org`.

[43] J. Gailly and M. Adler. zlib data compression library. `http://www.zlib.net`.

[44] H. Georgiadis and V. Vassalos. Improving the efficiency of xpath execution on relational systems. In *EDBT*, pages 570–587, 2006.

[45] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, pages 436–445, 1997.

[46] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[47] T. Grust. Accelerating xpath location steps. In *SIGMOD Conference*, pages 109–120, 2002.

[48] T. Grust, S. Sakr, and J. Teubner. Xquery on sql hosts. In *VLDB*, pages 252–263, 2004.

[49] T. Grust, M. van Keulen, and J. Teubner. Staircase join: Teach a relational dbms to watch its (axis) steps. In *VLDB*, pages 524–525, 2003.

[50] T. Grust, M. van Keulen, and J. Teubner. Accelerating xpath evaluation in any rdbms. *ACM Trans. Database Syst.*, 29:91–131, 2004.

[51] A. Halverson, J. Burger, L. Galanis, A. Kini, R. Krishnamurthy, A. N. Rao, F. Tian, S. Viglas, Y. Wang, J. F. Naughton, and D. J. DeWitt. Mixed mode xml query processing. In *VLDB*, pages 225–236, 2003.

[52] M. P. Haustein, T. Härder, C. Mathis, and M. W. 0002. Deweyids - the key to fine-grained management of xml documents. In *SBBD*, pages 85–99, 2005.

[53] S.-C. Haw and C.-S. Lee. Node labeling schemes in xml query optimization: A survey and trends. *IETE Tech Rev*, 26(2):88–100, 2009.

[54] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[55] H. V. Jagadish, S. Al-Khalifa, A. Chapman, L. V. S. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. Timber: A native xml database. *VLDB J.*, 11(4):274–291, 2002.

[56] H. Jiang, W. W. 0011, H. Lu, and J. X. Yu. Holistic twig joins on indexed xml documents. In *VLDB*, pages 273–284, 2003.

[57] H. Jiang, H. Lu, W. W. 0011, and B. C. Ooi. Xr-tree: Indexing xml data for efficient structural joins. In *ICDE*, pages 253–263, 2003.

[58] H. Jiang, H. Lu, W. Wang, and J. X. Yu. Path materialization revisited: An efficient storage model for xml data. In *Australasian Database Conference*, 2002.

[59] C.-C. Kanne and G. Moerkotte. Efficient storage of xml data. In *ICDE*, page 198, 2000.

[60] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *SIGMOD Conference*, pages 133–144, 2002.

[61] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *ICDE*, pages 129–140, 2002.

[62] M. Kay. XSL transformations (XSLT) version 2.0. W3C recommendation, W3C, Jan. 2007. http://www.w3.org/TR/2007/REC-xslt20-20070123/.

[63] Q. Li and B. Moon. Indexing and querying xml data for regular path expressions. In *VLDB*, pages 361–370, 2001.

[64] H. Liefke and D. Suciu. Xmill: An efficient compressor for xml data. In *SIGMOD Conference*, pages 153–164, 2000.

[65] J. Lu, T. Chen, and T. W. Ling. Efficient processing of xml twig patterns with parent child edges: a look-ahead approach. In *CIKM*, pages 533–542, 2004.

[66] J. Lu, T. W. Ling, C. Y. Chan, and T. Chen. From region encoding to extended dewey: On efficient processing of xml twig pattern matching. In *VLDB*, pages 193–204, 2005.

[67] J. Lu, T. W. Ling, T. Yu, C. Li, and W. Ni. Efficient processing of ordered xml twig pattern. In *DEXA*, pages 300–309, 2005.

[68] R. MacNicol and B. French. Sybase iq multiplex - designed for analytics. In *VLDB*, pages 1227–1230, 2004.

[69] M. Mann. Introduction to cataloging and the classification of books. 2008.

[70] A. Marian and J. Siméon. Projecting xml documents. In *VLDB*, pages 213–224, 2003.

[71] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[72] J. McHugh and J. Widom. Query optimization for xml. In *VLDB*, pages 315–326, 1999.

[73] J. McHugh, J. Widom, S. Abiteboul, Q. Luo, and A. Rajaraman. Indexing semistructured data. Technical report, Computer Science Dept., Stanford University, 1998.

[74] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, pages 277–295, 1999.

[75] J.-K. Min, M.-J. Park, and C.-W. Chung. Xpress: A queriable compression for xml data. In *SIGMOD Conference*, pages 122–133, 2003.

[76] M. M. Moro, Z. Vagena, and V. J. Tsotras. Tree-pattern queries on a lightweight xml processor. In *VLDB*, pages 205–216, 2005.

[77] J. F. Naughton, D. J. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, R. Ramamurthy, J. Shanmugasundaram, F. Tian, K. Tufte, S. Viglas, Y. Wang, C. Zhang, B. Jackson, A. Gupta, and R. Chen. The niagara internet query system. *IEEE Data Eng. Bull.*, 24(2):27–33, 2001.

[78] W. Ng, W. Y. Lam, and J. Cheng. Comparative analysis of xml compression technologies. *World Wide Web*, 9(1):5–33, 2006.

[79] W. Ng, W. Y. Lam, P. T. Wood, and M. Levene. Xcq: A queriable xml compression system. *Knowl. Inf. Syst.*, 10(4):421–452, 2006.

[80] G. Nicol, L. Wood, R. Sutor, V. Apparao, S. Isaacs, A. L. Hors, C. Wilson, M. Champion, J. Robie, S. Byrne, and I. Jacobs. Document object model (DOM) level 1 specification (second edition). W3C working draft, W3C, Sept. 2000. http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/.

[81] D. Olteanu, H. Meuss, T. Furche, and F. Bry. Xpath: Looking forward. In *EDBT Workshops*, pages 109–127, 2002.

[82] P. E. O'Neil, E. J. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. Ordpaths: Insert-friendly xml node labels. In *SIGMOD Conference*, pages 903–908, 2004.

[83] S. Paparizos, Y. Wu, L. V. S. Lakshmanan, and H. V. Jagadish. Tree logical classes for efficient evaluation of xquery. In *SIGMOD Conference*, pages 71–82, 2004.

[84] L. Qin, J. X. Yu, and B. Ding. *TwigList* : Make twig pattern matching fast. In *DASFAA*, pages 850–862, 2007.

[85] F. Rizzolo and A. O. Mendelzon. Indexing xml data with toxin. In *WebDB*, pages 49–54, 2001.

[86] S. Sakr. Xml compression techniques: A survey and comparison. *J. Comput. Syst. Sci.*, 75(5):303–322, 2009.

[87] A. Schmidt, M. L. Kersten, M. Windhouwer, and F. Waas. Efficient relational storage and retrieval of xml documents. In *WebDB (Selected Papers)*, pages 137–150, 2000.

[88] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. Xmark: A benchmark for xml data management. In *VLDB*, pages 974–985, 2002.

[89] J. Seward. The bzip2 and libbzip2 official home page. `http://www.bzip.org`.

[90] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as xml documents. *VLDB J.*, 10(2-3):133–154, 2001.

[91] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *VLDB*, pages 302–314, 1999.

[92] J. Siméon, D. Chamberlin, D. Florescu, S. Boag, M. F. Fernández, and J. Robie. XQuery 1.0: An XML query language. W3C recommendation, W3C, Jan. 2007. http://www.w3.org/TR/2007/REC-xquery-20070123/.

[93] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-store: A column-oriented dbms. In *VLDB*, pages 553–564, 2005.

[94] G. V. Subramanyam and P. S. Kumar. Efficient handling of sibling axis in xpath. In *COMAD*, pages 95–102, 2005.

[95] I. Tatarinov, S. Viglas, K. S. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang. Storing and querying ordered xml using a relational database system. In *SIGMOD Conference*, pages 204–215, 2002.

[96] P. M. Tolani and J. R. Haritsa. Xgrind: A query-friendly xml compressor. In *ICDE*, pages 225–234, 2002.

[97] Z. Vagena, N. Koudas, D. Srivastava, and V. J. Tsotras. Answering order-based queries over xml data. In *WWW (Special interest tracks and posters)*, pages 1162–1163, 2005.

[98] Z. Vagena, N. Koudas, D. Srivastava, and V. J. Tsotras. Efficient handling of positional predicates within xml query processing. In *XSym*, pages 68–83, 2005.

[99] P. Wadler. A formal semantics of patterns in xslt. In *In Markup Technologies*, pages 183–202. MIT Press, 1999.

[100] P. Wadler. Two semantics for xpath. Technical report, 2000.

[101] N. Walsh, M. Fernández, A. Malhotra, M. Nagy, and J. Marsh. XQuery 1.0 and XPath 2.0 data model (XDM). W3C recommendation, W3C, Jan. 2007. http://www.w3.org/TR/2007/REC-xpath-datamodel-20070123/.

[102] W. Wang, H. Wang, H. Lu, H. Jiang, X. Lin, and J. Li. Efficient processing of xml path queries using the disk-based f&b index. In *VLDB*, pages 145–156, 2005.

[103] Y. Wu, J. M. Patel, and H. V. Jagadish. Structural join order selection for xml query optimization. In *ICDE*, pages 443–454, 2003.

[104] M. Yoshikawa, T. Amagasa, T. Shimura, and S. Uemura. Xrel: a path-based approach to storage and retrieval of xml documents using relational databases. *ACM Trans. Internet Techn.*, 1(1):110–141, 2001.

[105] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD Conference*, pages 425–436, 2001.

[106] N. Zhang, V. Kacholia, and M. T. Özsu. A succinct physical storage scheme for efficient evaluation of path queries in xml. In *ICDE*, pages 54–65, 2004.

[107] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Trans. Inform. Theory*, 23(3):337–343, May 1977.